The zen of programming

Emilio Berti

September 29, 2021

Contents

1 Introduction **5**

1.1 General tips 5

1.1.1 How (not) to work with files 6

1.1.2 Global environment overflow 6

1.1.3 Nesting for/if chunks 6

1.1.4 Use functions for transferable, manageable code 8

2 R **11**

2.1 Readable code 11

2.1.1 Saving workspace image as *.RData* 11

3 python **13**

4 bash **15**

4.1 aliases 15

5 LaTeX **17**

Chapter 1

Introduction

I have been told that my programming skills are above average. Often, I am asked to develop or debug code and to explain why I coded scripts in the way I did. I realized that I follow a combination of personal rules-of-thumb, rules-of-thumb of other people who code better than me, and style guides from notorious people (e.g. Wickham tidyverse guide style) or big companies (e.g. Google's R Style Guide); they are doing better than me, and copying their style is probably a good idea.

In this booklet, I want to show some tips and tricks I follow routinely and to explain why I do some things in a certain way. I am sure better and more comprehensive guides already exist, but maybe not for people with ecological background, which is often fragmentary regarding coding.

1.1 General tips

These are my *laws* that I always try to follow. In some cases I violate some of them, e.g. in early-development or testing dozens of statistical models without clue of the underlying data. However, a final, releasable code should always follow all these laws. If I will release a code that does not follow one of these laws, I will be ashamed of myself – except in the case I want to prove a point on them. It is important to stress that these laws apply specifically to the scientific research environment and are not representative of how to properly code in other settings.

1. Data input, manipulation, and output must be explicit.

2. Do not overflow the (global) environment. Really guys, we are ecologists, be nice to the environment.

3. Do not nest more than three loops/conditional statements. If you did, rewrite everything from scratch.

4. If you're gonna do it twice, write a function for it.

These laws are fancy and general enough to be mis-understood. Let's expand on them.

1.1.1 How (not) to work with files

Nowadays, most of the analyses require large computations divided into steps. Intermediate output can be stored into files, which then can be used for downstream computations. A common mistake is to work with these intermediate files using point-and-click methods, often copy-pasting their content into scripts. This is an extremely bad practice for many reasons. First, there is no trace of what it has been done and from where the text in the script is coming from. Second, text editors often use special characters, e.g. linebreaks, that are not compatible within a script or among operating systems. Thirds, the potential for automation is severely reduced; for intsance, if you work with hundreds of files, the point-an-click steps need to be re-performed manually every time. Finally, the code is less readable, especially in the case the *csv* containing many rows.

A easy way to avoid all of this is to read the file in the code using reading functions, e.g. in `read.csv()`, `read_csv()`, or `fread` in *R*. This may seem basic, but it happens more than what I would like to admit.

1.1.2 Global environment overflow

I have seen many times a phenomenon that I call environment overflow, i.e. the (re)initialization of variables contained in a dataframe without deleting old copies. For example, a column (x) in a dataframe (*df*) can be extracted and passed to a new variable: `x <- df$x`. I consider this a bad practice because: first, it does not provide any new information; second, it created duplicates in the environment; and finally, it creates confusion for everyone (for instance, what's the difference between x and *d$x*?).

I have seen this used particularly when performing statistical tests or modelling. All main functions related to these accept a *data* argument, i.e. the dataframe where the variables are stored. So, why not to use directly this argument and avoid overflowing the workig environment? For instance, it is preferable to write

```
m <- lm(y ~ x, data = df)
```

instead of

```
x <- df$x
y <- df$y
m <- lm(y ~ x)
```

which takes more space and is less clear (where x and y are coming from), especially when there are multiple dataframes with the same variable names (is it x from *df1* or from *df2*?). Despite not overflowing directly the environment, also the `attach(df)` function in *R* generates similar confusions and should thus be avoided.

1.1.3 Nesting for/if chunks

Let's take a look at the following code comparing values from three vectors. Values are compared and then the relationships between them are reported.

```
x <- rnorm(100) #100 random normally distributed values
y <- rnorm(100)
z <- rnorm(100)
ans <- rep(NA, 100) #initialize answer
```

```
for (i in seq_along(x)) {
    if (x[i] > 0) {
        if (y[i] < z[i]) {
            if (y[i] < x[i]) {
                ans[i] <- "x > 0, x > y, y < z"
            } else {
                ans[i] <- "x > 0, x < y, y < z"
            }
        } else {
            ans[i] <- "x > 0, y > z"
        }
    } else {
        ans[i] <- "x < 0"
    }
}

ans[1:10]

 [1] "x < 0"              "x < 0"              "x > 0, y > z"
 [4] "x < 0"              "x > 0, x > y, y < z" "x < 0"
 [7] "x < 0"              "x < 0"              "x < 0"
[10] "x > 0, y > z"
```

The code above runs ok, performs the task it needs to do, but it can barely be read and understood. I can assure you that this is because there are four nested for/if statements. If you remove them, not only the code will be much readable, but, at least in *R*, it will also run faster. Let's try to rewrite it:

```
ans[x > 0 & x > y & y < z] <- "x > 0, x > y, y < z"
ans[x > 0 & x < y & y < z] <- "x > 0, x < y, y < z"
ans[x > 0 & y > z] <- "x > 0, y > z"
ans[x < 0] <- "x < 0"

ans[1:10]

 [1] "x < 0"              "x < 0"              "x > 0, y > z"
 [4] "x < 0"              "x > 0, x > y, y < z" "x < 0"
 [7] "x < 0"              "x < 0"              "x < 0"
[10] "x > 0, y > z"
```

It sure isn't pretty and it can still be improved, but just by removing the nested statements and using *R* native vectorized operator `&` we achieve the same task using four instead of 15 messy, unreadable lines. Also remember that in *R* vectorized operations are always the preferred native way of doing things, whereas for/if chuncks are quite slow and unefficient; we hit two birds with the same stone here.

1.1.4 Use functions for transferable, manageable code

Functions are you're biggest friends when you need to re-do the same tasks multiple times. In *R* functions are declated as:

```
my_fun <- function(arg1, arg2, ...) {
  # something to compute
  # . . .
  # something to return
}
```

where *my_fun* is the name of your function and *arg1* and *arg2* the arguments of the function. A simple function is the power of a number:

```
squared <- function(x) { #x is the number you want the power of
  ans <- x ** 2 #compute
  return(ans) #return
}

squared(2)

[1] 4
```

This function is quite useless, but it is useful to play with such useless functions to get a grasp on them. A more complex function can be to get the power of a number with random exponent between one and 10:

```
# compute the power *n* of a number,
# with *n* being randomly sampled between 1 and 10.
random_squared <- function(x) {
  root <- runif(1, 0.1, 1) * 10
  root <- round(root)
  ans <- x ** root
  message("The random exponent is: ", root)
  return(ans)
}

random_squared(1:5)

The random exponent is: 5
[1]    1   32  243 1024 3125
```

In *R* it is not necessary to return something and `return(x)` is the same as `x`. I learnt coding in *C*, where returns must be specified, and I prefer to explicitly write it. I couldn't find a negative consequence of explicitly returning the output, so I do it because it is more clear what it is returned.

Just to give an idea of how useful functions can be, let's take a look at a still relatively one I have used:

```
#' @title get correct UTM crs for the study area
#' @param df data.frame with "lon", "lat" coordinates.
```

```
3  #' @return crs in format "CRS" (sp package).
4  utm_crs <- function(df) {
5    if (!"lon" %in% colnames(df) | !"lat" %in% colnames(df)) {
6      stop("Missing 'lon' or 'lat' column")
7    }
8    lon <- df[, "lon"]
9    range_lon <- range(lon)
10   avg_lon <- mean(range_lon)
11   lat <- df[, "lat"]
12   range_lat <- range(lat)
13   avg_lat <- mean(range_lat)
14   utm <- floor((avg_lon + 180) / 6) + 1
15   epsg <- 32600 + utm
16   if (avg_lat < 0) {
17     epsg <- epsg + 100
18   }
19   ans <- raster::crs(paste0("EPSG:", epsg))
20   return(ans)
21 }
```

I did this because I wanted to obtain a UTM coordinate reference system from a lon-lat degree one. It is something that you can do every time you need it, but in this way I just write a separate file with this function that I call, without the need to copy-paste wildly. Also, if there is a mistake in the function (e.g. I should add 120 instead of 100 at line 17), I need to change this only once instead of several times in several scripts, with the risk that I forget to change it in all occurences, leading to error in the code.

Chapter 2

R

2.1 Readable code

Readability is almost everything. Rarely, a good, functioning code is written at the first attempt. Often, code written some time before need to be changed. If code is not readable, changes are difficult to implement. Therefore the question: how can we write readable code?

I will answer this question starting from another: for whom the code should be readable?

To researchers I suggest to write code as they write a manuscript for a scientific paper. Divide the code in sections as you would do with paragraphs. You may have, for example, a section to import all data, another to wrangle it, another to perform statistical analyses, etc... Treat each section as a paragraph. If a paragraph is very long, treat it as appendix material: put it in another script and call it where you need it.

1. Never save your workspace as *.RData*. If you need to save a *.RData* or *rds* data, explicitly save it.
2. Do not overflow the global environment

2.1.1 Saving workspace image as *.RData*

Chapter 3

python

Chapter 4

bash

4.1 aliases

When I started to need more sophisticated routines to run, I did what I would have done as if I was programming in *R*: I wrote a bash script and I called it, similar to a `source()` command in *R*. Sometimes this is the best way to do it, especially when arguments should be passed to script, but it often not the only or more elegant way to do it. For example, I wrote the following *GIServer.sh* script to connect to two servers with two IPs that were accessed passing an argument to the script (`$1`):

```
#!/bin/bash
 if [ $1 -eq 4 ]
then
        IP=00.00.0.000
 elif [ $1 -eq 1 ]
then
        IP=00.00.0.001
 else
        exit
 fi

rdesktop -d domain -g 2400x1300 -u emilioberti $IP
```

When I run `bash GIServer.sh 4`, a connection to the Windows server 4 was open in a window 2,400 × 1,300. Because I was working on several monitors at that time, and being lazy as I am, I originally wrote this with two extra arguments that would define the resolution of the window to be open. Then, I got tired of typing two extra arguments every time (I am lazy indeed) and I removed them, defining the resolution to be fixed at 2,400 × 1,300 that worked well for my main monitor. At this point, do I really need a script to do this?

The answer is 'no'. I need an *alias*. Aliases are variables that are assigned with a bash command and that can be accessed directly from the bash terminal. Basically, they are shortcuts to custom commands renamed as we want, and therefore easy to remember. An alias should be written in the *.bashrc* file, which is loaded at the user log-in. For example, I could replace the *GIServer.sh* script by writing two single lines in *.bashrc*:

```
alias gis4='rdesktop -d domain -g 2400x1300 -u emilioberti 00.00.0.000'
alias gis1='rdesktop -d domain -g 2400x1300 -u emilioberti 00.00.0.001'
```

At my next log-in (or sourcing *.bashrc* again: `source  /.bashrc`) I can just run `gis4` in the bash and get the same result as if I ran `bash GIServer.sh 4`.

Not always a bash script can be (or should be) substituted with an alias. However, when the bash script is for a simple task that it is supposed to be called frequently, using an alias instead of a script can be a good idea. In general, if the program does not need input arguments and you can write the code in one or few lines, writing an alias instead of bash script can be a good idea.

Chapter 5

$\LaTeX$