The zen of programming

Emilio Berti

September 29, 2021

Contents

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 $df\$x$?).

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 where needed, 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

If a code runs, good. If a code that runs is readable, great. 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?

There is a lot of emphasis in academia to learn how to write scientific papers for journals. 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

Bash is a Unix shell that provides command line user interface to the GNU/Linux operating system. Bash is one the main reason I prefer Linux over Windows. It comes with a pre-defined set of commands useful for job control and file and directory utilities. For instance, Bash makes it easy to locate files in the whole hard drive. For instance, the command to find a file containing the string *LICENSE* in its name is:

```
$ find . -maxdepth 2 -name '*LICENSE*'

./django-polls/LICENSE
./keras/LICENSE
./freetube/LICENSES.chromium.html
./freetube/LICENSE.electron.txt
./julia-1.5.2/LICENSE.md
./Downloads/LICENSES.chromium.html
./Downloads/LICENSE.electron.txt
```

the option `-maxdepth 2` limits the search within two children directories of the current location.

A comprehensive list of all useful commands is not in the scope of this guide, but the ones I use most often are:

`echo` prints strings on the terminal screen

`cd` changes directory

`pwd` prints the absolute path of the current directory

`mkdir` creates a directory

`touch` creates a file

`nano` starts the *nano* text editor in the terminal

`rm` removes files or directories

`ls` lists contents of the current directory

`grep` shows only files or strings containing a specific pattern

`cp` copies an existing files or directory to a new location

`mv` moves an existing files or directory to a new location

`tree` shows the directory tree of the current location

`ssh` connects via secure shell to remote machines

`history` shows the last commands run in Bash

`cat` prints out a single file or concatenate several ones

`head` prints the first lines of a file

`tail` prints the last lines of a file

`more` prints a file in the terminal with navigation control

`tr` removes or subtitute characters in a string or a file

`cut` separates a string or file according to a character and retrieve only specific columns

`chmod` administrates reading, writing, and executing priviledges of files

`ps` shows running processes

`kill` terminates processes

`git` for git version control

`zip/unzip` zips or unzips files

`wget` downloads stuff from internet

`curl` downloads stuff from internet

`man` shows the manual of a command

Pressing `Ctrl + r` starts a reverse search of the recently-used command. Command can be piped using `|`, where the output returned by the left expression is used as input by the right expression. For example `ls | grep *.pdf` will show only the files in the current directory that have *pdf* extension. This can be used to perform tasks that otherwise will require manual labour in a straightforward way. For intance, it happends quite often that we have multiple *csv* files that we want to concatenate (bind them row-wise) into one file. This can be done in other programming languages as *R* or *python*, but it is much easier (and faster) to do it in Bash:

```
find . -maxdepth 1 -name '*.csv' -print0 | xargs -0 cat > onefile.csv
```

The operator `>` redirect the output to *onefile.csv*, where the content of all csv files will be stored. At this point you may notice that the above code, when `-maxdepth 1` changes to other numers will not only concatenate files within the current directory, but also in all children directories depending on the number specified. The code above may look complicated, but once you get used to Bash it comes naturally to your mind, much before thinking of an alternative solution in *R*.

It may not be clear from this simple list why Bash is so powerful or what can be achieved by using it. But it is indeed the best companion to perform automated pipelines in a secure and scalable way. Bash is substantially an environment where it is possible to code in a programming language that is useful to perform operations on files or strings and to control processes and their flow. You can also add custom functionality specifying aliases (more about this below) and functions, most notably in the */.bashrc* file that is sourced when a Bash terminal is open. As an example consider the following function that I added to the */.bashrc* file:

```
uppercase() {
    echo $1 | tr '[:lower:]' '[:upper:]'
    echo $1 | tr '[:lower:]' '[:upper:]' | xclip -sel clip
}
$ uppercase 'hello world!'
HELLO WORLD!
```

In Bash, *$n* (where *n* is a number) means that that is an argument passed to the function. In the above code, the `uppercase()` function prints the passed argument (a string) and pass it (using `|`) to `tr` to replace lowercase characters with uppercase ones. The third line does the same thing but copies the output one the clipboard, so I can paste it using *Ctrl + v*.

4.1 aliases

Aliases renames existing command (or pipes of them) in one word that you find more familiar. For instance, I can never rememebr, so I added this to the */.bashrc*:

```
alias clip="xclip -selection c"
```

Instead of writing `xclip -selection c`, I can now only write `clip`, which will implicitly performs the same thing.

You can also write scripts that can be called in Bash. I wrote a script to get general information about the food additives *E###* from wikipedia. That's the code in the *wiki.sh* file:

```
#!/bin/bash

url=https://en.wikipedia.org/wiki/$1
file=/tmp/wiki.html
usagehtml=/tmp/tmp_use.html
usagetxt=/tmp/tmp_use.txt

curl -s $url -o $file #download wikipage into temporary folder
name=$(grep -i '<title>' $file | cut -d '>' -f 2 | cut -d '<' -f 1 | cut -d '-' -f 1)
grep 'used as' $file | grep food > $usagehtml
```

```
pandoc $usagehtml -o $usagetxt #use pandoc to convert html to txt
usage=$(grep 'used as' $usagetxt | cut -d '[' -f 2 | cut -d ']' -f 1)

rm $file $usagehtml $usagetxt #remove temporary files
echo $1, $name, $usage, $url #display results
```

This script can be called in bash running `$ bash wiki.sh` or by giving it running priviledges:

```
$ chmod +x wiki.sh
$ ./wiki.sh E150
E150, Caramel color , , https://en.wikipedia.org/wiki/E150
$ ./wiki.sh E214
E214, Ethylparaben , antifungal, https://en.wikipedia.org/wiki/E214
```