The zen of programming

Emilio Berti

October 18, 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

3 python **13**

4 Bash **15**

4.1 aliases 17

4.2 Bash scripts 17

5 Web servers **19**

5.1 Apache2 19

6 Spatial analysis **21**

6.1 R packages 21

6.2 python packages 22

6.3 Coordinate Reference System (CRS) 22

6.4 Spatial analysis in R 101 23

6.4.1 Reproject rasters and geometries 23

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-thumbs, rules-of-thumbs 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 guide, 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. The laws are:

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* contains many rows.

An easy way to avoid all of this is to read the file in the code using reading functions, e.g. `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 tasks 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 working 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 are x and y coming from?), especially when `lm()` is called many lines after the first two or 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()` 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 one I have used:

```
#' @title get correct UTM crs for the study area
#' @param df data.frame with "lon", "lat" coordinates.
#' @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 write down every time you need it, but by declaring the function I can call it 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 on learning how to write scientific papers for journals, but not on how to write proper code. To researchers, I suggest to write code as they would write a manuscript for a scientific paper. Divide the whole code into manageable stand-alone scripts that fit a purpose, e.g. data preparation (Introduction), analysis (Results), and visualization (Discussion). Divide each script in chunks as you would do with paragraphs, e.g. a first chunk to load the data, another one to explore it and apply necessary transformations, another one to save the transformed data into a new file. Treat each script and chunk as you would do with a manuscript section and paragraph: if a chunk is very long, split it (make a new paragraph); if a script is too long, split it into two (make a new section); if a chunk is not necessary for the main analysis, make a stand-alone script for it (move it to the appendix).

Some principles that I came up specifically for *R*:

1. Never let RStudio to save your workspace as *.RData*. If you want to save a *.RData* or *rds* data, save it explicitly. No data should be saved without users explicitly asking for it.

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 of the main reasons 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 `find` makes it easy to locate files in the whole hard drive. 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 commands.

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 have noticed 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. This is an example of an extremely tedious task that is made extremely easy in Bash. 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* (you will need to use at least the three functions `list.files()`, `read.csv()`, and `rbind()`).

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.

4.2 Bash scripts

You can also write scripts that can be called in Bash. As an example, below is a script I wrote to get general information about the food additives *E###* from wikipedia:

```
#!/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
```

I saved the script in a file called *wiki.sh* file, which 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
```

The line `#!/bin/bash` tells Bash which program to use to run the script, in this case Bash itself. If you want to run a *python* script *script.py* directly from Bash you can either do it by `$ python script.py` or by substituting `#!/bin/bash` with `#!/bin/python`, giving it running priviledges and run it with `$ ./script.py`.

Linux has a native way to schedule jobs, e.g. running a script. One of the most popular are *cron* and *crontab*. I'll not give further details here, but *crontab* provides an easy interface to schedule periodic jobs, e.g. uploading data to a remote location, running check-ups and updates, etc.

Chapter 5

Web servers

5.1 Apache2

Apache2 is a popular web server software to host a website. To install apache2 in Ubuntu:

```
$ sudo apt update
$ sudo apt install apache2
```

Sites are stored in the */var/www* folder.

```
$ sudo mkdir /var/www/gci
$ sudo nano /var/www/gci/index.html #write your html site
$ cd /etc/apache2/sites-available/
$ sudo cp 000-default.conf gci.conf
$ sudo nano gci.conf
####### in nano add ###################
# ServerAdmin emilio.berti90@gmail.com
# DocumentRoot /var/www/gci/
# ServerName gci.example.com
######################################
$ sudo a2ensite gci.conf
$ service apache2 reload
# sudo nano /etc/hosts
####### in nano add ###########
# 127.0.1.1     gci.example.com
####### in nano add ###########
```

Chapter 6

Spatial analysis

The most basic ideas of spatial analysis are **rasters** and **geometries**. A raster is a basically a matrix to which it has been associated several meta-data, defining for instance its geographic projection, resolution, etc. For most purposes, you should think to a raster as a pixel image with ancillary data to map it into a real place on Earth. Rasters can be stacked, similarly to multi-dimensional matrices (tensors); in R, these are called *RasterStack* (or bricks, I never understood the difference). To be stackable, rasters need to have the same geographic projection, resolution, and extent. Each raster in a stack is called a *layer* (sometimes also feature, especially in machine learning). Geometries are vector representations of spatial features. For example, a line is represented by a vector with origin, end, and module and are thus "scale-free". Rasters and geometries were developed for a common goal (mapping spatial features), but contrained by two different phylosophies, i.e. the necessity to map pixel-like features or continuous ones. If you are familiar with PDF file format, you should know that PDF encode, when possible, all information as vector geometries and, when it cannot, as pixel rasters. That's why you can zoom indefinitely in some PDFs and they still maintain the original resolution, but in others the image becomes blurry; you just hit the resolution limit of that pixel raster.

6.1 R packages

The world of spatial analysis is huge and scary. There are many packages to work with spatial stuff. In R, the most notable are **raster** (to be superseeded by the much faster **terra**) for raster analysis and **sp** for vector analysis. You need to install **gdal** (abstraction library) and **geos** (geometric representation) to use them (and **rgdal** and **rgeos** as well). Make sure you install all relevant libraries and packages, as some coordiante reference systems (more about this below) are contained only in one of them. I personally like also the **sf** package, a tidyverse substitute for *sp*. It is a more high-level interface to spatial geometries and it is thus easier to use than *sp*; it works great in conjuction with *dplyr* and, if you're using tidyverse, I strongly recommend to use *sf*. For low-level stuff, *sp* is probably more flexible and perhaps faster.

6.2 python packages

Will write this, for now: **shapely**, **fiona**, **GeoJSON**, and **GeoPandas**. Python is great to build pipelines, websites and visualizations. I cannot stress how much flexible and scalable python is in this regard.

6.3 Coordinate Reference System (CRS)

Rasters and geometries need a CRS to properly map their features in space. CRS tells us how to interpet pixels and vector in a precise spatial context. You can think to a CRS as ticks of xy axes of a scatterplot; without the ticks, we can see the points, but we have no information about how to interpet them. A CRS also defines the measurement unit, with the most commonly used being arc-degrees and meters.

CRS are commonly expressed as the *EPSG* code and by their *proj4string* expression. For most purposes, EPSG codes and proj4string are equivalent; EPSG (which stands for European Petroleum Survey Group) codes are IDs that refer to a specific proj4string. I find it easier to remember a four/five digit EPSG code than a complex proj4string, but that's me. Proj4string is, however, more explicit in its definition. A proj4string typically has many parameters, stored together as a string. For example:

+ init, an EPSG code (e.g. `+init=epsg:4326`)

+ proj, the projection used (e.g. `+proj=merc`)

+ ellps, the ellipsoid model of Earth (e.g. `+ellps=WGS84`)

+ units, the measurement units (e.g. `+units=m`)

A list of all parameters can be found at https://proj.org/ It is important to understand that all EPSG codes and proj4strings are community standards, i.e. definition of spatial coordinates that are useful for specific purposes. Nothing forbids you to define a custom proj4string, but, if it is really useful, it is probably already in the GDAL archive.

The website https://epsg.io/ contains all information you need to know about a particular CRS and how to convert it to another one. In R, you can get a list of all installed CRS with *rgdal::make_EPSG()*, which returns a data.frame with, among the others, the EPSG code and the proj4string of projection. As this is a data.frame, you can also search for a specific CRS. For instance, if you want only projections that include the string *berlin*, you can achieve it with the following code:

```
library(rgdal)
crs <- make_EPSG() #this has 6609 rows for me
berlin <- crs[grep('berlin', crs$note, ignore.case = TRUE),
              c("code", "prj4")] #only one for me
```

berlin is the CRS called Soldener Berlin *EPSG:3068*, with proj4 `+proj=cass +lat_0=52.4186482777778 +lon_0=13.6272036666667 +x_0=40000 +y_0=10000 +ellps=bessel +units=m +no_defs +type=crs:` https://epsg.io/3068. From the proj4string is clear that the unit of measurment is meters, a Bessel ellipsoid id used to approximate Earth surface, and the origin is at a longitude of 13.63 and latitude of 52.42. I was not aware of what a Bessel ellipsoid was, but a quick search susggests this is used for national surveys (which makes sense here for Germany) and that it will be replaced in the next decades by modern ellipsoids of satellite geodesy. This hints to a good point: standards change, so keep in mind that analyses made years ago may need to be adjusted for today's standards.

6.4 Spatial analysis in R 101

The raster I use is the estimated distribution of the African lion (*Panthera leo*) as it should be today without human interferences. I obtained this raster from PHYLACINE 1.2 database.

6.4.1 Reproject rasters and geometries

The first thing to know when re-projecting a layer is its original CRS. The only reason a layer does not already have a CRS is in the case you created it; in that case, you should specify the CRS of the layer yourself.

```
library(raster)

r <- raster("lion.gri")
crs(r)

CRS arguments:
 +proj=cea +lat_ts=30 +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m
+no_defs
```

The raster is in a Cylindrical Equal Area projection (*cea*). This is a quite good projection for some spatial analyses that focus on raster cells, as cea assures that each cell has the same area (not shape). However, I find it quite ugly to plot in this crs and I want re-project it to a Mollweide projection (Figure 6.1).

```
old_crs <- proj4string(crs(r))

reproj <- projectRaster(r,
                        crs = "+proj=moll +datum=WGS84 +units=m +no_defs",
                        method = "ngb") #nearest-neighbour
```

The proj4string of the new layer is `"+proj=moll +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs"`. I used a nearest-neighbour algorithm as this raster is binary (0 or 1) and that makes the most sense to me here.

The difference between the two layers projections are due to different CRS, but also approximations due to the algorithm chosen as method to resample the original raster. Different algorithm give different output (Figure 6.2) and you should choose which method is best for you.

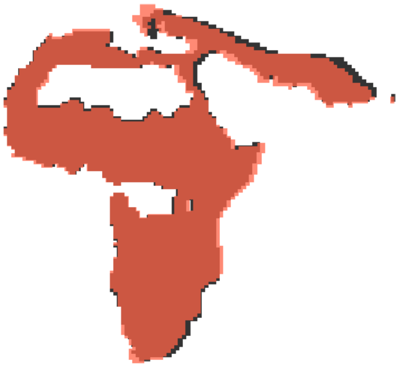

Figure 6.1: Present-natural distribution of the African lion in Berhman CEA projection (black) and Mollweide (red).

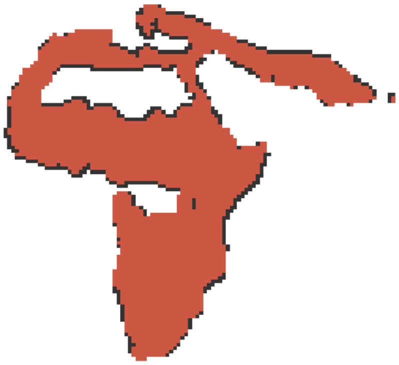

Figure 6.2: Present-natural distribution of the African lion in Mollweide projection using bilinear interpolation (black) and nearest-neighbour method (red).