Zero boilerplate

As a young man, I used to curate a collection of "template files", for all the formats and programming languages that I was using. I had a template for `LaTeX` files, a template for `C` programs, a template `html` page, etc. The basic idea was that *you never start writing a text file from scratch, you just copy an existing file and change some things.*

Now I realize that this is a bad idea. *You must start all your projects with an empty file.* This is the **zero boilerplate** idea. It is a liberating experience; you feel intimately connected to the underlying structure of the system that you rest upon. You grow roots, in a way.

Here I collect examples of complete files in various languages. Of course, you should never copy these files and edit them. Instead, you should memorize them an re-write the text each time than you need it.

1 HTML

In the old days, HTML was a very verbose language. With the advent of XHTML, the situation worsened so much that the default language of the web was impossible to write by humans. Fortunately, HTML5 corrects the errors of his predecessors and, thanks to default tagging, it is possible to create web documents by hand again.

This is a minimal **HTML** file

```
<!doctype html>
<title>Coco notes: zero boilerplate</title>

<h1>Zero boilerplate</h1>

<p>
Here begins the text of this page.
```

As advised by many people (e.g. the authors of HTML5, the google html coding guidelines), it is best to omit all implicit tags (such as `<html>` and `<head>` and `<body>`), and all automatically closed tags (such as `<p>`, `<li>` and many others).

If you want to use a tiny bit of css or mathjax (probably the only acceptable use of javascript), you can do this

```
<!doctype html>
```

```
<title>Coco pages: zero boilerplate</title>
<style type="text/css">
        body { max-width: 90ex; }
        pre { background:lightgray; width:80ch; }
</style>
<script src="//path.to/MathJax.js?config=TeX-AMS_CHTML-full" type="text/javascript"></script>


<h1>Zero boilerplate</h1>

<p>
Here begins the text of this page.
```

And, if you are a decent person, you should rarely need anything other than that.

2 TeX and LaTex

Plain **TeX** is notoriously boilerplate-free. You just write your paragraphs, separated by blank lines, and end your document with `\bye`. This is the best option to write a book, unless you rely on direct entry of UTF-8 characters (in which case you can use LuaTeX or XeTeX), fancier math (in which case you'll want AMS-TeX), or advanced sectioning and indexing capabilities (in which case you'll want LaTeX).

A minimal **LaTeX** file is thus

```
\documentclass{article}
\begin{document}
Hello, world!
\end{document}
```

and a LaTeX file in french and with some fancy math,

```
\documentclass{article}
\documentclass[a4paper,11pt]{article}  % classe article standard
\usepackage[utf8]{inputenc}            % pour écrire les accents directement
\usepackage[T1]{fontenc}               % pour faire des vrais guillemets
\usepackage[frenchb]{babel}            % conventions typographiques françaises
\usepackage{amsmath}                   % mathématiques avancées

\begin{document}
Voici le théorème de Stokes:
$$\int_{\partial\Omega}\omega=\int_\Omega\mathrm{d}\omega$$
\end{document}
```

you should strive to keep your list of packages minimal. The preamble to the document should *never* take more than half a screen.

3 C

The `c` compiler is always a good unix citizen. You can compile an empty file

```
cat /dev/null > x.c
cc -c x.c
```

and it will produce a valid empty object `x.o` that does not have any symbols (as you can verify, because `nm x.o` gives an empty output).

Thus, no boilerplate is needed for writing a `C` library.

If you want to write a command line program in **C**, it will look something like this:

```
#include <stdio.h>   // stderr, fprintf, printf
#include <stdlib.h>  // atof, free
#include "image.h"   // image_read, image_write

// regamma, a program to change the gamma of an image
int main(int c, char *v[])
{
        // extract input arguments
        if (c != 3)
                return fprintf(stderr, "usage:\n%s gamma in out\n", *v);
        double g      = atof(v[1]);
        char *filename_in  = v[2];
        char *filename_out = v[3];

        // read input images
        int w, h;
        float *x = image_read(filename_in, &w, &h);

        // change gamma in-place
        for (int i = 0; i < w*h; i++)
                x[i] = 255 * pow( x[i]/255 , g);

        // write output
        image_write(filename_out, x, w, h);

        // cleanup and exit
        free(x);
        return 0;
}
```

Notice that all the headers are explicitly justified to be necessary, by saying which functions are required from each header.

4 Python

Like `C`, the `Python` interpreter is also a good unix citizen. You can run the empty program, that does nothing and gives an empty output. Thus, there is no boilerplate required for python. If you want to use imports, this is easier than in `C` because the language can help you to specify which functions you need from each import.

```
from imageio              import imread, imwrite
from scipy.sparse         import eye, diags, kronsum
```

```
from scipy.sparse.linalg  import spsolve
from numpy                import clip, around, uint8
f = imread("lena.png")                          # read interior image
g = imread("landscape.png")                     # read background image
h,w = g.shape                                   # extract image dimensions
f = f.flatten()                                 # flatten image into a vector
g = g.flatten()                                 # flatten image into a vector
M = diags((g==0).astype(float))                 # mask operator (zero pixels of g)
x = eye(w, w, 1)                                # path graph of length W
y = eye(h, h, 1)                                # path graph of length H
G = kronsum(x,y) + kronsum(x,y).T               # kronecker sum (grid of size WxH)
L = G - diags(G.dot([1]*(w*h)))                 # laplacian operator
I = eye(w*h)                                    # identity operator
A =  I - M    - M*L                             # state the problem (operator)
b = (I - M)*g - M*L*f                           # state the problem (data)
x = spsolve(A, b)                               # solve the problem
imwrite("out.png", clip(around(x),0,255).astype(uint8).reshape(h,w)) # save x
```

Notice that you only import the strict minimum that you need.