



LINELENGTH:  
HERRAMIENTA Y ANÁLISIS DE CUMPLIMIENTO DE LA  
GUÍA DE ESTILO DE PYTHON EN RELACIÓN CON LA  
LONGITUD MÁXIMA DE LÍNEA

Curso Académico 2017/2018

Trabajo Fin de Grado

Autor : Kevin Oliva Muñoz

Tutor : Dr. Gregorio Robles



# Trabajo Fin de Grado

LINELENGTH: Herramienta y Análisis de Cumplimiento de la Guía de Estilo  
de Python en Relación con la Longitud Máxima de Línea

**Autor :** Kevin Oliva Muñoz

**Tutor :** Dr. Gregorio Robles Martínez

La defensa del presente Trabajo Fin de Grado se realizó el día            de  
de 2017, siendo calificada por el siguiente tribunal:

**Presidente:**

**Secretario:**

**Vocal:**

y habiendo obtenido la siguiente calificación:

**Calificación:**

Fuenlabrada, a            de            de 2017



*Dedicado a  
mi familia / mi abuelo / mi abuela*



# Agradecimientos

Aquí vienen los agradecimientos... Aunque está bien acordarse de la pareja, no hay que olvidarse de dar las gracias a tu madre, que aunque a veces no lo parezca disfrutará tanto de tus logros como tú... Además, la pareja quizás no sea para siempre, pero tu madre sí.





# Resumen

Los lenguajes de programación suelen tener una guía de estilo que indica, entre otras cuestiones, cómo ha de formatearse el código. Este proyecto pretende realizar un estudio sobre el uso de una de las reglas de estilo que contiene la guía de estilo de Python PEP8<sup>1</sup>. La regla en cuestión es la siguiente: “**Máxima longitud de las líneas:** Limita todas las líneas a un máximo de 79 caracteres”. Esta regla ha producido un pequeño debate en la comunidad Python, ya que seguirla al pie de la letra muchas veces tiene como resultado un peor código. Esto es debido a que los desarrolladores acortan, por ejemplo, el nombre de las variables hasta hacerlas ininteligibles.

En el estudio realizado en este proyecto intentamos detectar las líneas de código que han sufrido esta mala práctica por parte de los desarrolladores. Así, intentamos identificar cuándo los desarrolladores han cambiado el nombre de variables, dándoles un nombre más corto, para cuadrar la longitud de la línea a 79 caracteres. Si esto ocurre nos encontramos ante un problema, ya que la regla de estilo de máxima longitud está produciendo un efecto contrario al que pretendía.

Para realizar este estudio se ha implementado el lado del servidor con Node.js (utilizando Express) y el lenguaje JavaScript.

---

<sup>1</sup><http://https://pythonwiki.wikispaces.com/pep8>



# Summary

This project is a study on a PEP8 rule. This rule is “Maximum line length: Limits all lines to a maximum of 79 characters”. This rule has caused a discussion in the python community. The question is ¿Is it possible to leave a worse code when using the pep8 rules?. This study tries to find the lines that have undergone a bad practice. This practice is change the name of a variable to a shorter one so that the line has 79. Now the name of the variable is worse than before and our code is unreadable, when this happens, the PEP rule has gotten the opposite of what it wanted. This project does a python code analysis in github, depends on the results of the analysis, we will decide if this bad practice is widespread in python developers.



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	1
1.2. La Guía de Estilo PEP8 . . . . .	3
1.3. Estructura de la memoria . . . . .	4
<b>2. Objetivos</b>	<b>7</b>
2.1. Objetivo principal . . . . .	7
2.2. Objetivos secundarios . . . . .	8
<b>3. Estado del arte</b>	<b>11</b>
3.1. App Cliente-Servidor . . . . .	11
3.2. Python . . . . .	12
3.3. <code>pycodestyle</code> (antes <code>pep8</code> ) . . . . .	13
3.4. Otros programas del estilo de <code>pycodestyle</code> . . . . .	14
3.5. JavaScript . . . . .	15
3.6. Node JS . . . . .	16
3.7. Express . . . . .	17
3.8. MongoDB . . . . .	18
3.9. JSON . . . . .	18
3.10. Git . . . . .	19
3.11. GitFlow . . . . .	21
<b>4. Diseño e implementación</b>	<b>25</b>
4.1. Análisis de los repositorios . . . . .	25
4.2. Análisis del grupo de líneas seleccionadas para nuestro estudio . . . . .	29

4.3. Ejecución de la instrucción <code>git log</code> . . . . .	29
4.4. Análisis del pasado de una línea . . . . .	33
4.5. Clasificación de líneas conflictivas . . . . .	35
<b>5. Resultados</b>	<b>37</b>
5.1. Resultados del análisis general . . . . .	37
5.2. Resultados del análisis del grupo conflictivo . . . . .	39
<b>6. Conclusiones</b>	<b>43</b>
6.1. Consecución de objetivos . . . . .	43
6.2. Aplicación de lo aprendido . . . . .	45
6.3. Lecciones aprendidas . . . . .	46
6.4. Trabajos futuros . . . . .	46
6.5. Valoración personal . . . . .	47
<b>A. Instalaciones necesarias</b>	<b>49</b>
A.1. NodeJS . . . . .	49
A.2. NPM . . . . .	50
A.3. Creación de un nuevo proyecto Express . . . . .	51
<b>Bibliografía</b>	<b>53</b>

# Índice de figuras

3.1. Esquema básico de la estructura cliente-servidor . . . . .	12
3.2. Python . . . . .	12
3.3. JavaScript . . . . .	15
3.4. Node.js . . . . .	16
3.5. Express.js . . . . .	17
3.6. MongoDB . . . . .	18
3.7. Estructura de aplicación MEAN: Mongo + Express + Angular + Node . . . . .	19
3.8. Ejemplo del formato JSON . . . . .	20
3.9. Esquema de como organizan la información los demás VCS . . . . .	20
3.10. Esquema de como git organiza la información . . . . .	21
3.11. Diagrama GitFlow . . . . .	22
3.12. Vemos como podemos iniciar el método GitFlow en nuestro proyecto de manera sencilla. . . . .	24
3.13. Imágenes de ejemplo de uso de Source Tree . . . . .	24
4.1. Diagrama de flujo del funcionamiento del estudio . . . . .	26
4.2. Salida por defecto de la instrucción git log . . . . .	30
4.3. Salida de la instrucción git log con opciones . . . . .	31
A.1. Descarga de NodeJS . . . . .	49
A.2. Formato del package.json después de lanzar el comando npm init . . . . .	50
A.3. Formato nuestro package.json . . . . .	51
A.4. Estructura básica de un proyecto . . . . .	52





# Capítulo 1

## Introducción

### 1.1. Motivación

Cuando terminas de estudiar y por fin empiezas a dedicarte profesionalmente al desarrollo de software, rápidamente te das cuenta de una cosa, que nunca, o casi nunca, habías pensado en tu vida de estudiante: en tu trabajo pasas más tiempo leyendo código de otra(s) persona(s) que tu propio código. Esto hace que, repentinamente, empiecen a cobrar sentido las frases que tanto te repetían tus profesores sobre estructurar bien tu código o elegir nombres “inteligentes” para las variables o funciones.

Coincidiendo con el inicio de mi etapa profesional, cuando verdaderamente me he dado cuenta de la importancia de escribir código de calidad, y no sólo “algo que funcione”, mi tutor, Gregorio me habló de un vídeo con una charla en una *PyCon*, el congreso anual más grande sobre el lenguaje Python, en el que Raymond Hettinger, uno de los desarrolladores principales de Python, expone buenas prácticas para escribir un código legible y de calidad<sup>1</sup>. En el vídeo se indica que se ha de seguir unas normas de estilo (que vienen recogidas en la guía de estilo de Python, conocida como PEP8<sup>2</sup>, ya que eso facilita la tarea de comprensión del código. Sin embargo, en el vídeo se plantea lo que para mí es una interesante cuestión: ¿Podemos dejar un código peor de lo que estaba al realizar un cambio para cumplir con una regla de la guía de estilo de Python?

---

<sup>1</sup><https://www.youtube.com/watch?v=wf-BqAjZb8M>

<sup>2</sup>PEP es el acrónimo de Python Enhancement Proposal, un documento que se propone y debate en la comunidad Python antes de ser definitivamente aceptado como “oficial”, que es cuando se le asigna finalmente un número.

Dicha cuestión es la que vamos a estudiar en este TFG. Nos centraremos en un caso muy concreto, a la vez que muy debatido en la comunidad de Python. La regla en cuestión es la siguiente:

“Máxima longitud de las líneas: Limita todas las líneas a un máximo de 79 caracteres”

El caso que nos preocupa es el siguiente: un desarrollador encuentra ante una línea de código mayor a los 79 caracteres, por lo que decide cambiarla, acortando para ello el nombre de alguna variable. Al hacerlo, el nombre de la variable carece de sentido, por lo que la legibilidad del código se resiente.

Para entender mejor esto que explicamos, vamos a poner un pequeño ejemplo. Imaginemos que el desarrollador ha escrito la siguiente línea de código:

```
'subtitles': self.extract-subtitles(video_id, video_subtitles_id)
```

Si estamos leyendo código y pasamos por esta línea, se entiende bastante bien, sin necesidad de contar con más contexto. Se puede suponer fácilmente que estamos obteniendo los subtítulos, los cuales obtenemos de una función llamada “extract-subtitles”. El nombre de la función parece que se ha elegido con sentido, al ser informativo. También sabemos que para ello le pasamos como parámetros el id del vídeo y el id de los subtítulos.

Si estamos analizando código y tenemos todo el contexto de nuestro programa, es posible que no haga falta ir a la documentación de la función “extract-subtitles”, porque sabemos todo lo que nos hace falta con esta línea y podremos seguir leyendo el código que estamos analizando sin perder más tiempo.

Esta línea en sí, no ocupa más que 69 caracteres. Pero, supongamos que por motivos de *indentación*<sup>3</sup>, esta línea superara los 79 caracteres. El desarrollador ha intentado refactorizar su código pero no consigue quitar las *indentaciones* y como le ha saltado el *warning* de que no cumple la regla de estilo únicamente por la longitud de la línea, decide para no perder más tiempo, acortar el nombre de las variables, y forzando un caso extremo, también el de la función, de tal forma que la línea quedara con menos de 80 caracteres y cumplir con la regla. De forma que la línea resultante podría quedar de la siguiente forma:

---

<sup>3</sup>Este término *indentación* significa mover un bloque de texto hacia la derecha insertando espacios o tabuladores, para así separarlo del margen izquierdo y mejor distinguirlo del texto adyacente. Es utilizado para mejorar la legibilidad del código fuente por parte de los programadores.

```
'subtitles': self.e_s(v_id, v_s_id)
```

Como se puede ver, la línea ahora es más difícil de explicar de por sí. Si estamos analizando código bien puede pasar que seríamos incapaces de saber cómo estamos obteniendo la variable “subtitles”, necesitaríamos irnos a la función para intentar entender qué hace, y qué parámetros le están llegando. La línea ahora cumple las reglas de estilo, pero es ilegible. Hemos ido a peor.

Este hecho me parece digno de estudio, por varios motivos. Uno de ellos, indudablemente, está relacionado con el debate que puede crear en la comunidad Python, sobre si un código que cumple la guía de estilo PEP8 puede ser de peor calidad que uno que no lo cumpla.

Pero siendo sincero, el motivo por el cual decidí realizar el proyecto sobre este tema fue que, con mi experiencia como estudiante, no pude evitar imaginar el siguiente pensamiento por parte de cualquiera de nosotros: “He terminado mi práctica de Python, voy a pasar a mi código un filtro que me he descargado de la guía de estilo PEP8 para entregar un buen código, lo más importante es que pase este filtro. Por tanto, si me salta algún error haré cualquier cambio rápido y sin pensar mucho sólo para forzar a que el filtro pase correctamente.”

Creo firmemente que, como estudiantes, no somos capaces de asimilar la importancia de escribir un código de calidad, y de las horas y horas que ahorraremos a nuestros compañeros de trabajo, o a nosotros mismo cuando retomamos código propio de hace tiempo. Son cosas simples, pero es importante que desde el principio de nuestra formación se adopten estas buenas costumbres.

## 1.2. La Guía de Estilo PEP8

Desde que empezamos a escribir código y conforme progresamos, adquirimos ciertas pautas que definen la manera en que expresamos nuestras ideas en el lenguaje que manejemos, ya sea JavaScript, Python, C u otro. Esto define nuestro estilo, y tiene en cuenta la manera en la que *indentamos*, definimos nombres de variables y funciones, o el proceso de realizar o no comentarios de código, entre otros.

¿Por qué es tan importante seguir un estilo cuando estamos programando? El principal y más importante es porque el código debe ser mantenido, ya sea por nosotros mismos o por otras personas. Y a lo largo de nuestra vida como desarrolladores, pasamos muchas más horas leyendo código que escribiéndolo.

Para esto nacieron las guías de estilo de código. El principal objetivo de una guía de estilo es que nuestro código sea más fácil de leer, compartir y analizar. Facilita la consistencia entre el código fuente de distintos usuarios y hace que nuestro código sea mucho más mantenible.

Las guías de estilo son algo que está generalizado en los distintos lenguajes de programación, por ejemplo para PHP tenemos PSR-2<sup>4</sup>, y en JavaScript una de las más notables es Crockford<sup>5</sup>. No hay una sola guía de estilo para cada lenguaje, podemos encontrar varias de distintas fuentes (es muy común que los proyectos de software libre tengan una propia). Algo muy común es consultar las guías que ha creado Google. Por ejemplo, para Java podemos encontrar la siguiente<sup>6</sup>.

En esta memoria nos centraremos en la guía de estilo para Python PEP8. La guía de estilo PEP8 fue realizada por Guido van Rossum, Barry Warsaw y Nick Coghlan. Está dedicada a la recopilación de los estándares seguidos por los desarrolladores de Python a la hora de escribir código Python para la librería estándar. Esta guía está conformada por varias convenciones, en las cuales no entraremos en profundidad, destacaremos una de ellas, alrededor de la cual se basa este proyecto: “Limitar los tamaños de línea a 79 caracteres como máximo”.

Para aplicar las reglas de calidad que marca PEP8, tenemos a nuestra disposición varias herramientas automáticas para pasarle a nuestro código. En el capítulo 3 hablaremos en profundidad sobre una de ellas, esta herramienta es `pycodestyle`, antes conocida como `pep8`.

### 1.3. Estructura de la memoria

Creemos conveniente explicar muy brevemente la estructura de la memoria, dando a conocer los objetivos de cada capítulo para facilitar así la lectura de la misma:

1. **Introducción:** En este capítulo se explica el contexto de este proyecto, así como las razones por las que se elige el tema a tratar.

---

<sup>4</sup><http://www.php-fig.org/psr/psr-2/>

<sup>5</sup><http://javascript.crockford.com/code.html>

<sup>6</sup><https://google.github.io/styleguide/javaguide.html>

2. **Objetivos:** En este capítulo se detalla cada uno de los objetivos que se han planteado desde el inicio del proyecto.
3. **Estado del arte:** Aquí presentamos las tecnologías con las que se ha implementado el proyecto, además de algún concepto para entender mejor la estructura del mismo.
4. **Diseño e implementación:** Este capítulo profundiza e intenta explicar en detalle el desarrollo del proyecto.
5. **Resultados:** Aquí se presenta, a modo de resumen, como ha quedado finalmente el proyecto.
6. **Conclusiones:** Es el capítulo final; en él se intenta evaluar de forma general el proyecto, haciendo hincapié en los conceptos aprendidos en su elaboración.

Finalmente se expondrá la bibliografía, que se ha consultado para la elaboración del trabajo y de la memoria.



# Capítulo 2

## Objetivos

### 2.1. Objetivo principal

Este proyecto tiene como objetivo principal:

“Analizar una serie de repositorios reales de GitHub con bastante código Python, para obtener unos resultados y en función de ellos, decidir si el hábito explicado en el apartado anterior es un problema real en los desarrolladores de Python”.

El proyecto consistirá en un programa software que analizará código fuente de un programa en el lenguaje de programación Python y detectará aquellas líneas que han sido acortadas cambiando el nombre de una variable por un nombre más corto, y por tanto, menos entendible, para conseguir pasar la regla de la guía de estilo PEP8 de la máxima longitud de línea.

Por una parte se pretende hacer un estudio bastante amplio, con un volumen de repositorios de GitHub grande, para obtener unos resultados lo suficientemente amplios como para poder llegar a una conclusión. Se busca detectar si este mal hábito se encuentra extendido entre los desarrolladores de Python.

Por otra parte, también se pretende alcanzar otro objetivo más a largo alcance, el cual consiste en intentar interiorizar una cuestión, sobre todo a nivel de estudiantes. Dicha cuestión es la importancia de saber realmente lo que es un buen código, sobre todo, no caer en el típico error cuando eres estudiante de pensar que un buen código es algo que funciona.

Según está orientado en la actualidad el aprendizaje de la programación a nivel de universidad, es muy difícil darse cuenta que el código que produces no es algo que vas a escribir una

vez, vas a probar que funciona, vas a entregar y no vas a volver a ver en tu vida. Este ciclo es lo que suele pasar con el código que creas como estudiante. Esto unido a ciertas circunstancias, como agobios y prisas por los plazos de entrega, o que en algunas ocasiones sólo se valore el resultado del programa, hace que generalmente se creen malos hábitos por parte de los alumnos a la hora de programar.

El proyecto intenta llegar a este objetivo creando un “debate” alrededor del caso concreto que se estudia. ¿Es mejor el código que no cumple la guía de estilo PEP8 antes de realizar el cambio, pero que tiene un buen nombre de variable?, ¿o es mejor el código después del cambio que cumple la guía de estilo PEP8 pero que ha dejado la variable con un nombre sin sentido?

Quizás la respuesta correcta sea decir que se debe cambiar la línea de código para que cumpla la regla de máxima longitud de línea pero realizando otro cambio, y si no es posible cambiar esa línea, darle una vuelta al código para *refactorizarlo* y hacer que todas las líneas cumplan de esta forma las reglas de calidad.

Lo que se intenta decir con esto es que cuando un estudiante ya ha adquirido ciertos conocimientos técnicos sobre la programación y sobre la generación de código de calidad, debe darse cuenta que tenemos en nuestra mano una serie de reglas de calidad como la guía de estilo PEP8 o similares, las cuales se deben utilizar, pero no de cualquier forma. Es decir, hay que pararse a pensar un minuto y ser coherente e inteligente a la hora de aplicarlas en nuestro desarrollo, y no dejar un código peor que el que teníamos antes por intentar forzar a que pasen estas reglas de cualquier manera.

## 2.2. Objetivos secundarios

Aparte del objetivo explicado anteriormente, a la hora de realizar este proyecto, se han tenido en cuenta los siguientes objetivos secundarios:

- **Trabajar de una forma orientada al mundo profesional:** Se ha intentado llevar una metodología de trabajo lo más parecida posible a lo que nos encontraremos en un futuro entorno profesional. Para ello se ha intentado mantener un uso constante de `git` y `GitFlow`.
- **Aplicar tecnologías e ideas no vistas durante la vida universitaria:** El proyecto de fin de carrera es el broche, la guinda a nuestra vida universitaria. Debido a esto, he intentado



basar este proyecto sobre tecnologías e ideas de trabajo que no conozco en profundidad, ya que no se han enseñado de refilón en mi vida universitaria. De esta forma lo que busco es ampliar mis conocimientos y aprovechar esto para conocer nuevas herramientas que me pueden valer para mi futura vida laboral. Esta cuestión puede dividirse, sobre todo, en dos casos:

1. En cuanto a la aplicación de tecnologías, he usado un servidor JavaScript, Node.js con Express. Mi elección se debe a que en la universidad se ha enseñado a montar servidores en lenguajes como Java, y sobre todo, con Python y Django. Para la base de datos he utilizado MongoDB, en lugar de SQL por el mismo motivo. En la universidad hemos utilizado bases de datos relacionales, y quería utilizar una no relacional, para entender mejor los *pros* y *contras* de cada una de ellas, y tener más conocimientos sobre cuál elegir en mis futuros desarrollos.
2. Enfocar una idea de trabajo más orientada a la investigación. Normalmente en nuestra vida como estudiante partimos siempre de un enunciado redactado al empezar a desarrollar. En el proyecto tenemos la oportunidad de investigar sobre una idea para realizar luego nuestro desarrollo.



# Capítulo 3

## Estado del arte

En esta sección vamos a describir brevemente las tecnologías aplicadas en o relacionadas con este proyecto.

### 3.1. App Cliente-Servidor

La arquitectura *cliente-servidor* es una de las más extendidas en la actualidad. Dicha estructura es un modelo de aplicación distribuida en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados **servidores**, y los demandantes, llamados **clientes** (véase figura 3.1). Un cliente realiza peticiones a otro programa, y es el servidor quien atiende su petición y le da respuesta:

Esta arquitectura presenta una clara separación de las responsabilidades, lo que facilita y clarifica el diseño del sistema. Es una arquitectura claramente centralizada, ya que toda la información reside en el servidor, y el cliente es el que realiza peticiones para obtener dicha información. Esto provoca que el servidor posea una lógica más compleja, y que potencialmente sea capaz de manejar de forma distinta las peticiones dependiendo de quién las realice.

En nuestro proyecto no hemos realizado como tal una *app* de cliente-servidor, ya que ha ido más enfocado a la investigación y a la obtención de un resultado. Sin embargo, el código realizado para obtener dichos resultados se ha dejado organizado en la parte del servidor de una aplicación web, para obtenerlo con distintas peticiones. Si en un futuro se quiere mejorar este proyecto, y se decide hacer una aplicación web con el contenido investigado aquí, solo sería necesario realizar la parte del cliente.

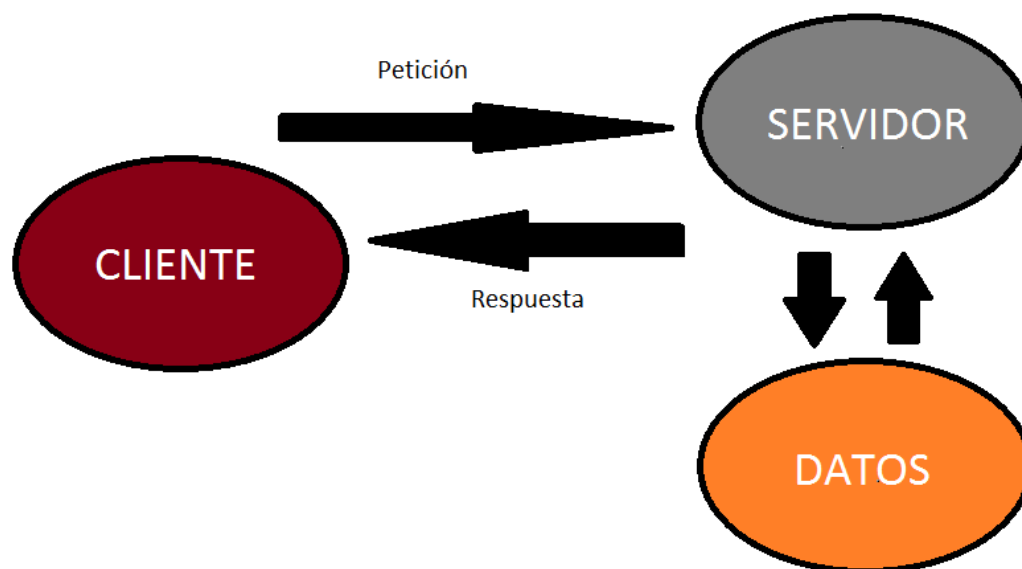


Figura 3.1: Esquema básico de la estructura cliente-servidor



Figura 3.2: Python

## 3.2. Python

Python es un lenguaje de programación de muy alto nivel, multiplataforma y multiparadigma, el cual, fue creado a finales de los años 80 por el holandés Guido van Rossum, Python fue creado como sucesor del lenguaje de programación ABC. El logo de Python se puede ver en la figura 3.2.

¿Qué quiere decir que es un lenguaje multiplataforma? Quiere decir que es un lenguaje interpretado, y por tanto no necesita compilación. Esto tiene como ventaja, por ejemplo, que da igual el sistema operativo donde se ejecute, o que aporta mayor rapidez al desarrollarlo.

¿Qué quiere decir que es multiparadigma? Esto quiere decir que no obliga al desarrollador a adaptarse a un estilo de programación concreto, Python soporta programación funcional, programación imperativa y programación orientada a objetos.

Python se caracteriza por tener una sintaxis limpia y ordenada, haciendo mucho hincapié en la legibilidad del código, a pesar de ello, tiene funcionalidades muy avanzadas, propias de lengua-

jes como C o C++.

Python es un lenguaje que ha adquirido gran popularidad en los últimos años, algunas de las razones por las que esto ha sucedido son las siguientes:

- Sencillez y velocidad a la hora de crear desarrollos completos, normalmente un programa en Python tendrá muchas menos líneas que sus equivalente en otros lenguajes (java, C, etc).
- Ofrece una gran cantidad de librerías, con funciones incorporadas, funcionalidades o tipos de datos, que favorecen la realización de muchas tareas sin necesidad de empezar completamente de 0.
- Se puede obtener de forma gratuita y utilizarse con fines comerciales.

### 3.3. pycodestyle (antes pep8)

Como ya hemos dicho anteriormente cuando hablábamos de PEP8, existen herramientas con las que podemos comprobar de forma automática si nuestro código cumple las reglas que dicta el PEP8. quí vamos a hablar un poco más en profundidad de una de ellas: “pycodestyle”, antes conocida como “PEP8”. Se cambió de nombre para evitar confusiones con la propia guía de estilo.

Esta herramienta nos permite pasar un análisis una vez nuestro código está terminado, para ver si cumple calidad. Podemos encontrar información sobre esta herramienta en GitHub<sup>1</sup>. Básicamente, la forma de utilizarla y lo que nos aporta es lo siguiente:

Podemos instalar, actualizar o desinstalar la herramienta de forma fácil con los siguientes comandos:

```
$ pip install pycodestyle
$ pip install -- upgrade pycodestyle
$ pip uninstall pycodestyle
```

---

<sup>1</sup><https://github.com/PyCQA/pycodestyle>

Una vez instalado podemos revisar nuestro código de forma sencilla, por ejemplo, con el siguiente comando, analizamos el código fuente del fichero “optparse.py”:

```
$ pycodestyle -- first optparse.py
```

La salida que `pycodestyle` nos proporciona sería la siguiente:

```
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '&#39;(&#39;
optparse.py:357:17: E201 whitespace after '&#39;{&#39;
optparse.py:425:80: E501 line too long (82 characters)
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use '&#39;in&#39;
```

Como vemos, `pycodestyle` nos muestra de manera clara la línea y el sitio justo donde estamos cometiendo el error de calidad y por el cual no cumplimos la norma correspondiente de la guía PEP8.

Podemos ver por ejemplo como en la siguiente línea

```
optparse.py:425:80: E501 line too long (82 characters)
```

se nos muestra el mensaje del error que un programador verá cuando una línea supera el máximo tamaño de línea permitido. Esta herramienta detecta el error, pero es el desarrollador es el que debe tomar la decisión de cómo solucionarlo.

### 3.4. Otros programas del estilo de `pycodestyle`

A parte del actual `pycodestyle`, existen otras herramientas parecidas que también nos sirven para analizar la calidad de nuestro código Python. Entre ellas cabe destacar `pylint` y `pyflakes`.



Figura 3.3: JavaScript

- **pyflakes:** Es un programa sencillo que comprueba los errores de los archivos fuente de Python. Funciona analizando el archivo de origen, no importándolo, por lo que es seguro utilizar en módulos con efectos secundarios. También es mucho más rápido.
- **pylint:** Es otro comprobador de código fuente para Python. Sigue el estilo recomendado por PEP 8. Es similar a `pychecker` y `pyflakes`, pero incluye las siguientes características: comprobación de la longitud de cada línea, comprobar si los nombres de las variables están bien formados de acuerdo con la norma de codificación del proyecto, comprobación de si las interfaces declaradas se implementan realmente.

## 3.5. JavaScript

JavaScript es un lenguaje orientado a objetos y basado en prototipos. Una de sus características más importantes es que se define como un lenguaje asíncrono, por lo que es muy útil para reaccionar a eventos por parte del usuario. Es un lenguaje interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios. En la figura 3.3 se puede ver el logo de JavaScript.

Se utiliza principalmente en su forma del lado del cliente (*client-side*), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas, una página web dinámica es aquella que incorpora efectos como texto que aparece y desaparece, animaciones, acciones que se activan al pulsar botones y ventanas con mensajes de aviso al usuario.

Sin embargo, existe una forma de JavaScript del lado del servidor(*Server-side JavaScript* o



Figura 3.4: Node.js

SSJS), como veremos más adelante en esta memoria cuando profundicemos en Node.js.

### 3.6. Node JS

Node.js es una librería y entorno de ejecución de E/S dirigida por eventos y por lo tanto, asíncrona que se ejecuta sobre el intérprete de JavaScript creado por Google V8. El logo de Node.js se muestra en la figura 3.4.

Node.js ejecuta JavaScript utilizando el motor V8, desarrollado por Google. Este motor permite a Node.js proporcionar un entorno de ejecución del lado del servidor que compila y ejecuta JavaScript a velocidades increíbles. El aumento de velocidad es importante debido a que V8 compila JavaScript en código de máquina nativo, en lugar de interpretarlo o ejecutarlo como *bytecode*. Node es de código abierto, y se ejecuta en Mac OS X, Windows y Linux.

Aunque JavaScript tradicionalmente ha sido relegado a realizar tareas menores en el navegador, es actualmente un lenguaje de programación total, tan capaz como cualquier otro lenguaje tradicional como C++, Ruby o Java. Además JavaScript tiene la ventaja de poseer un excelente modelo de eventos, ideal para la programación asíncrona.

¿Cuál es el problema con los programas de servidor actuales? A medida que crece su base de clientes, si quieres que tu aplicación soporte más usuarios, necesitarás agregar más y más servidores. Esto hace que el cuello de botella en toda la arquitectura de aplicación Web era el número máximo de conexiones concurrentes que podía manejar un servidor. Node.js resuelve este problema cambiando la forma en que se realiza una conexión con el servidor. En lugar de generar un nuevo hilo de OS para cada conexión (y de asignarle la memoria acompañante), cada conexión dispara una ejecución de evento dentro del proceso del motor de Node.js. Node.js también afirma que nunca se quedará en punto muerto, porque no se permiten bloqueos.





Figura 3.5: Express.js

## 3.7. Express

Express es sin duda el framework más conocido de Node.js, es una extensión del poderoso connect<sup>2</sup> y está inspirado en Sinatra<sup>3</sup>, además es robusto, rápido, flexible, simple, etc. Proporciona una delgada capa de características de aplicación web básicas, que no ocultan las características de Node.js que tanto ama y conoce.

El verdadero éxito de Express se encuentra en lo sencillo que es de usar. Tienes la capacidad de crear de forma rápida y sencilla una API sólida. Además, Express abarca un sin número de aspectos que muchos desconocen pero son necesarios. El logo de Express se puede ver en la figura 3.5.

De entre las tantas cosas que tiene este framework podemos destacar:

- Session Handler.
- 11 *middleware* poderosos así como de terceros.
- cookieParser, bodyParser . . .
- vhost
- router



Figura 3.6: MongoDB

### 3.8. MongoDB

MongoDB forma parte de la nueva familia de sistemas de base de datos NoSQL, es la base de datos NoSQL líder y permite a las empresas ser más ágiles y escalables. El logo de MongoDB se muestra en la figura 3.6.

En lugar de guardar los datos en tablas como se hace en las base de datos relacionales, MongoDB guarda estructuras de datos en documentos similares a JSON con un esquema dinámico (MongoDB utiliza una especificación llamada BSON). Esto hace que sea una base de datos ágil ya que permite a los esquemas cambiar rápidamente cuando las aplicaciones evolucionan, proporcionando siempre la funcionalidad que los desarrolladores esperan de las bases de datos tradicionales, tales como índices secundarios, un lenguaje completo de búsquedas y consistencia estricta.

La arquitectura completa de las tecnologías utilizadas en este proyecto se puede ver en la figura 3.7.

### 3.9. JSON

JSON (JavaScript Object Notation) es un formato de texto ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript aunque hoy, debido a su amplia adopción como alternativa a XML, se considera un formato de lenguaje independiente. JSON nació como una alternativa a XML, el fácil uso en JavaScript ha generado

---

<sup>2</sup><https://github.com/senchalabs/connect#readme>

<sup>3</sup><http://www.sinatrarb.com/documentation.html>

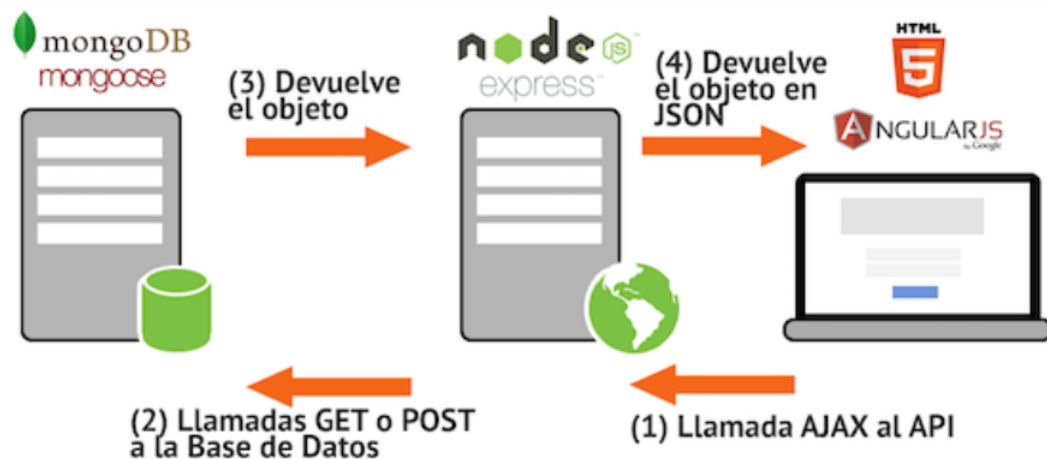


Figura 3.7: Estructura de aplicación MEAN: Mongo + Express + Angular + Node

un gran número de seguidores de esta alternativa. Se puede ver un ejemplo de una estructura de datos en JSON en la figura 3.8.

Una de las supuestas ventajas de JSON sobre XML como formato de intercambio de datos es que es mucho más sencillo escribir un analizador sintáctico (parser) de JSON, lo cual ha sido fundamental para que JSON haya sido aceptado por parte de la comunidad de desarrolladores AJAX, debido a la ubicuidad de JavaScript en casi cualquier navegador web.

Otra ventaja a tener en cuenta del uso de JSON es que puede ser leído por cualquier lenguaje de programación. Por lo tanto, puede ser usado para el intercambio de información entre distintas tecnologías.

## 3.10. Git

Los sistemas de control de versiones (VCS) son programas que tienen como objetivo controlar los cambios en el desarrollo de cualquier tipo de software, permitiendo conocer el estado actual de un proyecto, los cambios que se le han realizado a cualquiera de sus piezas, las personas que intervinieron en ellos, etc.

`git` es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. La principal diferencia entre `git` y cualquier otro VCS es cómo `git` modela sus datos. Conceptualmente, la mayoría de los demás sistemas

```

{ "users":[
  {
    "firstName":"Ray",
    "lastName":"Villalobos",
    "joined": {
      "month":"January",
      "day":12,
      "year":2012
    }
  },
  {
    "firstName":"John",
    "lastName":"Jones",
    "joined": {
      "month":"April",
      "day":28,
      "year":2010
    }
  }
]}

```

Figura 3.8: Ejemplo del formato JSON

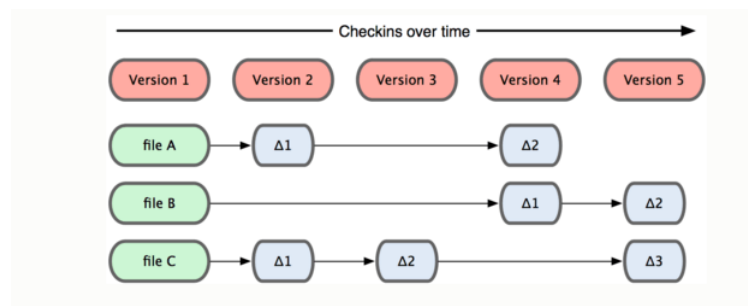


Figura 3.9: Esquema de como organizan la información los demás VCS

almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo, como se puede ver en la figura 3.9.

Como se puede ver en la figura 3.10, `git` no modela ni almacena sus datos de este modo. En cambio, `git` modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, `git` no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado.

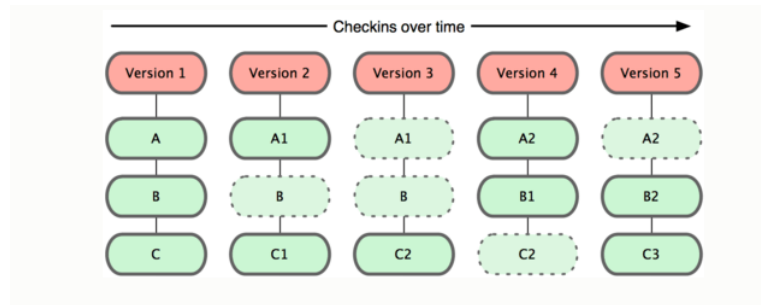


Figura 3.10: Esquema de como `git` organiza la información

## 3.11. GitFlow

GitFlow es un modelo de flujo de trabajo para `git` que da muchísima importancia a las ramas y que de hecho las crea de varios tipos, de forma temática, tal que cada tipo de rama es creada con un objetivo en concreto, como se puede ver en la figura 3.11:

1. **Master:** La rama *master* es la única rama existente que nos proporciona `git` al crear un repositorio nuevo. Esta rama tiene como objetivo ser el contenido del servidor de producción. Es decir, el HEAD de esta rama ha de apuntar en todo momento a la última versión de nuestro proyecto.

No se va a desarrollar desde esta rama en ningún momento.

2. **Develop:** Esta rama funciona paralelamente a la *master*. Si la anterior contenía las versiones desplegadas en producción, esta (que también estará sincronizada con `origin/develop`) contendrá el último estado de nuestro proyecto. Es decir, esta rama contiene todo el desarrollo del proyecto hasta el último `commit` realizado.

Cuando esta rama adquiera estabilidad y los desarrolladores quieran lanzar una nueva versión, bastará con hacer un *merge* a la rama *master*. Esto será lo que cree una nueva versión de nuestro proyecto.

3. **Features:** Que varias personas trabajen sobre la misma rama es bastante caótico ya que se aumenta el número de conflictos que se dan. A pesar de que los repositorios distribuidos



Figura 3.11: Diagrama GitFlow

faciliten esta tarea al guardar los `commits` solo localmente, tiene mucho más sentido usar la potencia de las ramas de `git`.

Cada vez que necesitemos programar una nueva característica en nuestro proyecto crearemos una nueva rama para la tarea.

Ya que *develop* contiene la última foto de nuestro proyecto, crearemos la nueva rama a partir de aquí. Una vez finalizada la tarea, solo tendremos que integrar la rama creada dentro de *develop*. Una vez integrada la rama en *develop*, podremos eliminarla y actualizar *origin*.

4. **Release:** Cuando hemos decidido que el código desarrollado hasta ahora pertenece a una versión de nuestro proyecto, y tenemos actualizado la rama *develop* con dicho código, crearemos una rama *release*.
5. **HotFix:** Cuando detectamos algún error en producción (rama *master*), creamos una rama *HotFix*, en la cual se arregla rápidamente el error (sin hacer ningún desarrollo más). Cuando tenemos el arreglo preparado integramos esta rama con la rama *master* y la *develop* para que todos los desarrolladores tengan el error solventado.

En el caso de este proyecto, es difícil usar todo el esquema explicado aquí de GITFLOW, debido a que es un proyecto pequeño, es decir, sin distintos entornos de desarrollo (desarrollo, preproducción, producción, etc). Además sólo lo ha desarrollado una persona a la vez. Sin embargo, he visto muy útil la utilización de la rama *feature*, para tener siempre la última versión

estable guardada en *develop*, y aislar el desarrollo de nuevas funcionalidades en éstas ramas. Una vez terminada y probada la nueva funcionalidad, integraba la rama *feature* con la *develop* y borraba la *feature*. Este ha sido el uso que he dado en mi caso a `GitFlow`. Para llevar esto a cabo, he encontrado *Source Tree* (véanse la figura 3.12 y la figura 3.13), que es una interfaz visual para utilizar Git. Es especialmente útil cuando queremos utilizar `GitFlow`, ya que es un poco lioso de entender con tantas ramas, sobre todo cuando se mantiene todo de forma manual. Con *Source Tree* podremos ver todo el esquema de ramas que forme nuestro proyecto de forma visual y crear las distintas ramas de forma mucho más fácil.

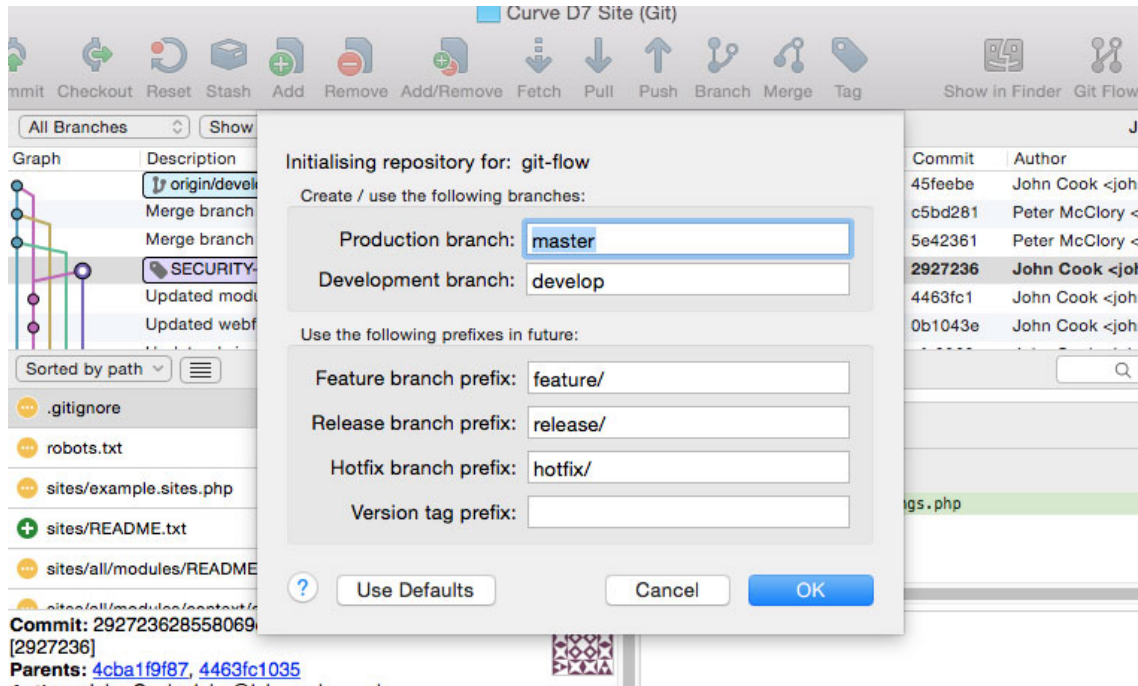


Figura 3.12: Vemos como podemos iniciar el método GitFlow en nuestro proyecto de manera sencilla.

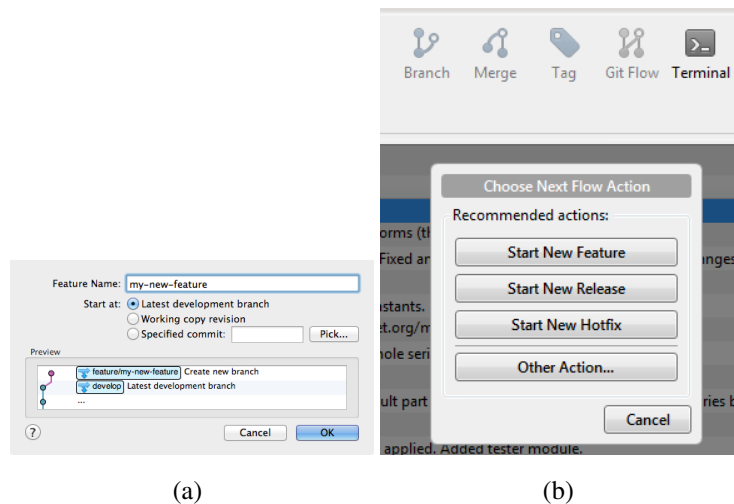


Figura 3.13: Imágenes de ejemplo de uso de Source Tree



# Capítulo 4

## Diseño e implementación

En el diagrama de flujo de la figura 4.1 se muestra el funcionamiento de nuestro estudio sobre el problema que estamos tratando en este proyecto. En las siguientes secciones se entrará en detalle sobre cada uno de los pases.

### 4.1. Análisis de los repositorios

Lo primero que hacemos es analizar una serie de repositorios que tenemos preparados para realizar el estudio. Cuando nos encontremos con un fichero Python y analicemos sus líneas, las dividiremos en tres grupos, dependiendo de la longitud que poseen en la actualidad:

- Grupo 1: “Zona verde”, la línea posee menos de 70 caracteres de longitud.
- Grupo 2: “Zona naranja”, la línea posee entre 70 y 79 caracteres de longitud. Este grupo de líneas es el que nos va a interesar, ya que tienen la posibilidad de haber estado en la “Zona roja” (veremos en el siguiente grupo) en algún momento de su historia, y en la actualidad estar en la zona naranja, por un cambio de una variable.
- Grupo 3: “Zona roja”, la línea posee 80 o más caracteres de longitud. Este grupo de líneas no cumple con las reglas de estilo PEP8.

Cuando nos encontramos una línea que pertenece a la zona naranja, lo que haremos es crear un nuevo objeto en nuestra base de datos, en nuestro esquema Linea. El objeto que guardaremos

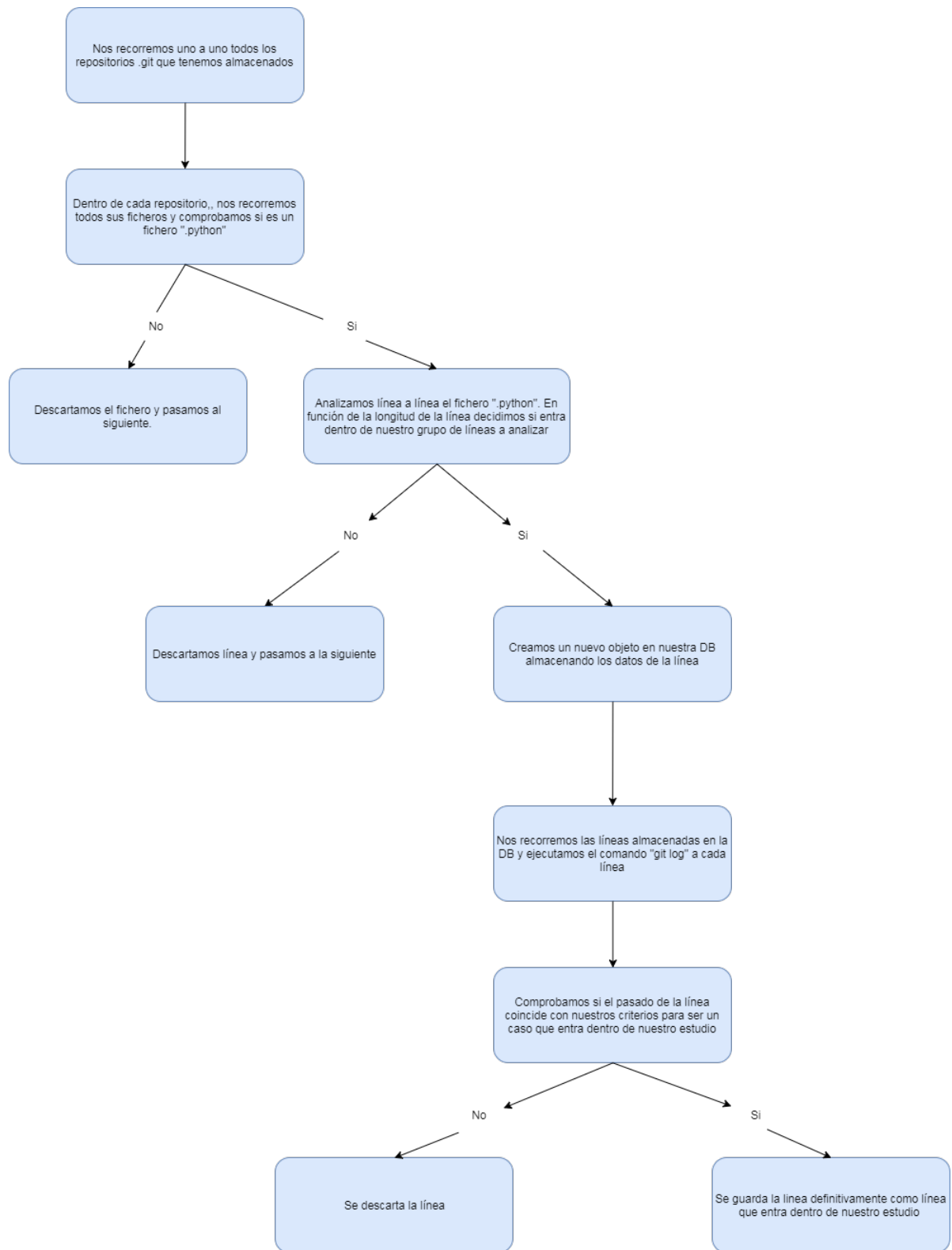


Figura 4.1: Diagrama de flujo del funcionamiento del estudio

tiene el siguiente aspecto:

```
var lineaSchema = mongoose.Schema({  
  
  linea: String,  
  fichero: String,  
  numLinea: String,  
  longitudLinea: String,  
  pasadoRojo: Boolean  
  historia: []  
  
});
```

Donde:

- **línea:** será la línea en sí, su contenido.
- **fichero:** será la ruta donde encontraremos el fichero dentro del repositorio que estemos analizando.
- **numLínea:** será el número que tiene esa línea dentro del fichero.
- **longitudLinea:** será la longitud de la línea.
- **pasadoRojo:** En este punto siempre iniciaremos a *False* este parámetro, ya que todavía no hemos comprobado el pasado de la línea. Lo haremos más adelante.
- **historia:** En este punto, siempre inicializaremos un *array* vacío en este parámetro. Al igual que el anterior, realizaremos esta comprobación más adelante. Cuando lo hagamos, lo que guardaremos aquí serán las distintas versiones de la línea en las que su longitud era mayor de 80 caracteres. Más adelante veremos con más detalle las diferentes comprobaciones que hacemos para actualizar este parámetro.

Además de esto, llevaremos la cuenta del total de líneas que nos hemos ido encontrando de cada grupo en el análisis conjunto de todos los repositorios, para al final mostrar los datos analizados organizados, y poder sacar una buena conclusión sobre el estudio. De esta forma, al terminar de analizar todos los repositorios crearemos un nuevo objeto en nuestra base de datos, con el siguiente formato:

```
var repositorioSchema = mongoose.Schema({  
  
  nombreRepo: Array,  
  lineasVerdes: String,  
  lineasRojas: String,  
  lineasNaranjas: String  
  
});
```

Donde:

- **nombreRepo:** Será un *array* que posea el nombre de todos los repositorios que hemos analizado.
- **lineasVerdes:** Será el número total de líneas pertenecientes a este grupo que hemos encontrado en nuestro análisis.
- **lineasRojas:** Será el número total de líneas pertenecientes a este grupo que hemos encontrado en nuestro análisis
- **lineasNaranjas:** Será el número total de líneas pertenecientes a este grupo que hemos encontrado en nuestro análisis.

## 4.2. Análisis del grupo de líneas seleccionadas para nuestro estudio

Lo primero que hacemos es recorrer todas las líneas que tenemos guardadas en la base de datos. Como hemos visto anteriormente, en este punto tendremos sólo las líneas que pertenecen al grupo de la “zona naranja”. Por tanto, recorreremos todas las líneas para analizarlas.

La primera comprobación que haremos es mirar si la línea es un comentario, y no estamos ante una línea de código real. Si nos encontramos ante un comentario descartaremos la línea. En el caso contrario, lo que haremos es ejecutar la instrucción `git log` a dicha línea para estudiar su pasado.

## 4.3. Ejecución de la instrucción `git log`.

En primer lugar, vamos a poner un poco en contexto: lo que buscamos es saber todo el historial de cambios que ha sufrido una línea en concreto dentro de un fichero.

Para llevar a cabo esto, en un principio estudiamos la posibilidad de utilizar el comando `git blame`, que da información sobre la historia de una línea. `blame` se suele utilizar en `git` para depurar errores, sobre todo. Este comando nos permite filtrar por unas determinadas líneas dentro de un fichero, podemos ver el *commit* que modificó por última vez cada una de las líneas. Sin embargo, la información que proporciona `blame` se nos quedaba un poco escaso para nuestro propósito, por lo que, terminando de revisar otras funcionalidades de `git`, y también revisando las distintas librerías JavaScript que nos permitieran implementar esto mediante código en nuestro servidor, tomamos la decisión de explotar la instrucción `git log`.

Vamos a ver en qué consiste la instrucción `git log`, y las opciones que nosotros hemos utilizado. Si ejecutamos la instrucción `git log` sin ningún parámetro, por defecto lo que nos devuelve es una lista de todos los *commits* realizados en ese repositorio en orden cronológico inverso. Es decir, muestra primero los *commits* más recientes. En la figura 4.2 podemos observar el formato en que `git log` nos da información.

Lo que nos muestra, como podemos ver, es cada *commit* con su suma de comprobación

```

$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

```

Figura 4.2: Salida por defecto de la instrucción `git log`

SHA-1, el nombre y correo del autor, la fecha y el mensaje que el autor ha escrito para ese *commit*.

La instrucción `git log` posee múltiples opciones para ajustar la búsqueda que realmente queremos, la opción que nosotros utilizamos fue la siguiente:

```
git log -L numeroLinea,numeroLinea:nombreFichero
```

De esta forma obtendremos la lista de todos los cambios que ha sufrido la línea que indiquemos desde su creación en el fichero hasta la actualidad.

Podemos ver en el siguiente ejemplo, la salida del comando que estamos tratando en nuestro código se muestra en la figura 4.3.

```
git log -L 155,155:git-web--browse.sh
```

Analizando esta salida, podemos ver que la línea 155 del fichero `git-web--browse.sh` ha sufrido los siguientes cambios.

1. Cuando la línea se incluyó dentro del fichero por primera vez en un `commit` fue como:

```
firefox|iceweasel)
```

2. El primer cambio que sufrió fue borrar las tabulaciones del principio:

```
$ git log --pretty=short -u -L 155,155:git-web--browse.sh
commit 81f42f11496b9117273939c98d270af273c8a463
Author: Giuseppe Bilotta <giuseppe.bilotta@gmail.com>

    web--browse: support opera, seamonkey and elinks

diff --git a/git-web--browse.sh b/git-web--browse.sh
--- a/git-web--browse.sh
+++ b/git-web--browse.sh
@@ -143,1 +143,1 @@
-firefox|iceweasel)
+firefox|iceweasel|seamonkey|iceape)

commit a180055a47c6793eaaba6289f623cff32644215b
Author: Giuseppe Bilotta <giuseppe.bilotta@gmail.com>

    web--browse: coding style

diff --git a/git-web--browse.sh b/git-web--browse.sh
--- a/git-web--browse.sh
+++ b/git-web--browse.sh
@@ -142,1 +142,1 @@
-    firefox|iceweasel)
+firefox|iceweasel)

commit 5884f1fe96b33d9666a78e660042b1e3e5f9f4d9
Author: Christian Couder <chriscool@tuxfamily.org>

    Rename 'git-help--browse.sh' to 'git-web--browse.sh'.

diff --git a/git-web--browse.sh b/git-web--browse.sh
--- /dev/null
+++ b/git-web--browse.sh
@@ -0,0 +127,1 @@
+    firefox|iceweasel)
```

Figura 4.3: Salida de la instrucción `git log` con opciones

```
firefox|iceweasel)
```

3. Y por último se le añadió texto:

```
firefox|iceweasel|seamonkey|iceape)
```

Para poder llevar todo esto a cabo en nuestro código hemos utilizado la librería de JavaScript `simple-git`<sup>1</sup>.

Siguiendo ahora con el análisis de nuestro proceso, recordemos que tenemos guardadas en nuestra base de datos las líneas que hemos detectado como posibles conflictivas, y que queremos analizar, guardadas con el siguiente formato:

```
var lineaSchema = mongoose.Schema({  
  
  linea: String,  
  fichero: String,  
  numLinea: String,  
  longitudLinea: String,  
  pasadoRojo: Boolean  
  historia: []  
  
});
```

Lo que hacemos en este punto es ejecutar la instrucción `git log` (con las opciones que acabamos de explicar), y en el caso de que nuestra línea haya tenido en algún momento de su historia una longitud mayor a 80, modificaremos su registro en la base de datos cambiando el valor del parámetro “pasadoRojo” a `True` y también nos guardaremos todo el pasado de la línea para analizarlo.

---

<sup>1</sup><https://www.npmjs.com/package/simple-git>



## 4.4. Análisis del pasado de una línea

Como hemos ido explicando durante esta memoria, lo que nosotros estamos buscando exactamente en este punto, es la iteración en la que la línea ha pasado de estar en la zona roja (mayor de 80 caracteres) a estar en la zona naranja (entre 70 y 79 caracteres).

Muchas veces, se puede dar esta iteración y aun así no ser el caso que incumbe a nuestro estudio, por tanto, debemos analizar el pasado de la línea, y para implementarlo en código, la única forma que tenemos es comprobar que en ese cambio, la línea actual y la línea en el pasado son suficientemente parecidas como para que el cambio pueda deberse a nuestro motivo de estudio, es decir, que la línea actual haya pasado de tener más de 80 caracteres, a tener entre 70 y 79, y que el cambio haya sido acortar una el nombre de una variable.

Vamos a intentar clarificar este último párrafo viendo esta cuestión con algún ejemplo. Lo que queremos decir es que la línea puede tener infinitos cambios en su pasado, de hecho puede ser que pase de la zona roja a la zona naranja, vuelva otra vez a la zona roja, pase a la verde, etc. Lo que queremos comprobar es que la línea actual es lo suficientemente parecida a la última versión de la línea en la que tenía más de 80 caracteres, para comprobar como ya explicamos en el párrafo anterior si el cambio se produce por nuestro motivo. Si vemos el siguiente ejemplo sacado de nuestro propio estudio:

```
settings = {'FEED_FORMAT': format, 'FEED_EXPORT_INDENT': None}
settings = {'FEED_FORMAT': format, 'FEED_EXPORT_INDENT_WIDTH': None}
```

La primera línea de código, es la que hay en la actualidad en el repositorio, y la segunda es la última versión que tenía más de 80 caracteres en su pasado. Es decir, esta línea entraría claramente en nuestro análisis, ya que podemos comprobar la línea en la actualidad es prácticamente igual a la versión de la línea que tenía más de 80 caracteres, de hecho, podemos intuir que la línea no ha sufrido más cambios después, y que la iteración de zona roja a zona naranja es el último cambio sufrido por dicha línea.

Sin embargo, imaginemos que la línea ha sufrido ese cambio, pero más adelante sufre otros, de forma que la línea actual queda bastante cambiada, pero sigue teniendo las características de

estar en la zona naranja, la comparativa entre la última versión de la línea en la zona roja y la línea actual quedaría así.

```
[settings, data] = {'FEED_FORMAT': None}
settings = {'FEED_FORMAT': format, 'FEED_EXPORT_INDENT_WIDTH': None}
```

Esta línea no entraría seguramente en nuestros filtros, ya que como podemos ver la línea en la actualidad no es lo suficientemente parecida a su última versión en la zona roja. Esta línea ha pasado de estar en la zona roja a la zona naranja a lo largo de su historia, sin embargo, no entraría dentro de nuestros parámetros.

Como decimos, en este punto todavía no podemos saber si la línea ha sufrido ese cambio por nuestro motivo, sin embargo, lo que sí podemos filtrar es que la línea es suficientemente parecida a su pasado como para que pueda darse, y para ello vamos a utilizar la distancia de Levenshtein<sup>2</sup>.

Teníamos guardadas todo el pasado de la línea, lo que haremos es comparar cada estado de la línea en su pasado, con el estado actual, y basándonos en la respuesta que nos devuelva la distancia de Levenshtein tomar una decisión. Si la respuesta está dentro de nuestro umbral, nos quedaremos con “ese pasado de la línea”, en caso contrario, lo desecharemos.

Al terminar este proceso,, tendremos un *array* con todos los estados de la línea en su pasado que han sido mayores de 80 y que se parecen lo suficiente a la línea actual como para ser nuestro caso. En este *array* normalmente sólo tendremos un registro, que será el paso anterior a cómo era la línea justo antes de cómo es en la actualidad. En caso de que más estados de la línea en su pasado entren en estos criterios y se cuelen aquí, solo interesaría el último caso en que la línea ha estado en la zona roja antes de pasar a la naranja, por tanto, más adelante los filtraremos.

En este punto volvemos a actualizar el objeto que tenemos para dicha línea guardado en nuestra base de datos. En este caso modificaremos el parámetro “historia:[]”, dándole como valor el *array* que comentamos en el párrafo anterior.

---

<sup>2</sup>[https://es.wikipedia.org/wiki/Distancia\\_de\\_Levenshtein](https://es.wikipedia.org/wiki/Distancia_de_Levenshtein)

Una vez hecho todo esto, limpiaremos la base de datos, de forma que borraremos todos los registros de las líneas que no hayan cumplido alguna de estas comprobaciones.

## 4.5. Clasificación de líneas conflictivas

En este punto, tenemos en nuestra base de datos únicamente las líneas que consideramos conflictivas para nuestro caso de estudio, después de pasar por todas nuestras comprobaciones. Ahora lo que necesitamos es clasificar estas líneas según creamos que es la razón por la que esa línea se ha acortado y ha pasado de nuestra zona roja a nuestra zona naranja.

Dentro de los posibles casos, hemos decidido clasificar las líneas en 4 grupos –recordemos que siempre comparando la iteración en la que la línea pasa de tener más de 80 caracteres a tener entre 70 y 79–:

1. La línea se diferencia por el único cambio de una palabra que ha pasado a ser más corta (caso que buscamos).
2. La línea se diferencia por que se han eliminado 4 espacios (tabulaciones).
3. La línea se diferencia porque se han eliminado 4 espacios (tabulaciones), y además ha cambiado una sola palabra, la cual se ha acortado (posible caso buscado).
4. Resto de líneas.

En resumen, lo que hemos hecho en código para implementar esto es lo siguiente: Iremos pasando cada proceso que vamos a explicar a continuación a cada grupo de líneas (línea antigua con más de 80 y línea actual entre 70 y 79), si la línea cumple con algún proceso se añadirá a dicho grupo, si la línea no encaja en ningún proceso se añadirá al último grupo **Resto de líneas**.

- **La línea se diferencia por el único cambio de una palabra que ha pasado a ser más corta:** Partimos tanto la línea antigua como la línea actual por espacios, de manera que nos quedamos con dos arrays los cuales guardan las palabras de cada línea. Comparamos estos dos arrays, y si sólo se diferencian en una posición, querrá decir que solo hay una palabra distinta. Esta palabra puede que sea una variable, por tanto se añade a este grupo.

- **La línea se diferencia por que se han eliminado 4 espacios:** Lo que hacemos es eliminar los 4 primeros caracteres de la línea antigua, y guardarnos el resto. Comparamos el resto de la línea antigua con la línea actual, y comprobamos si ahora son exactamente iguales, y además si los 4 caracteres que hemos eliminado de la línea antigua son 4 espacios. Si esto se cumple se añade a este grupo.
- **La línea se diferencia porque se han eliminado 4 espacios (tabulaciones), y además ha cambiado una sola palabra, la cual se ha acortado:** Lo que hacemos aquí es comparar si la línea antigua y la actual son distintas en un principio, pero luego, eliminando los espacios y tabulaciones, tienen el mismo número de palabras, de esta forma, sabremos que solo se diferencian por espacios. Después de esto comprobamos si todas las palabras son exactamente iguales menos una. Si esto ocurre, estamos ante una línea de este grupo.

Hemos elegido estos cuatro casos porque son los más comunes, por ejemplo, el caso de las tabulaciones, y los casos que afectan directamente a nuestro estudio, como es el 1 principalmente, y como puede ser el 3. De esta forma ya tenemos los resultados obtenidos por nuestro estudio.

# Capítulo 5

## Resultados

### 5.1. Resultados del análisis general

Una vez que hemos creado el software de apoyo, tenemos ante nosotros la posibilidad de analizar un gran volumen de líneas de código en lenguaje Python.

No presentamos una aplicación que ayude al desarrollador al mismo tiempo que está programando, o que le permite revisarlo una vez terminado, como podrían ser otros proyectos sobre el análisis de la calidad del código como pueden ser `pycodestyle`, `pyflakes` o `pylint`. Más bien, lo que ofrecemos es un análisis de un amplio volumen de código, un estudio, para intentar llegar a una conclusión sobre un problema planteado.

Cabe destacar que para realizar nuestro estudio hemos intentado recoger datos lo más cercanos posible al código que hay en el “mundo real” y que nos permitan sacar conclusiones reales.

Los datos que hemos utilizado para analizar son repositorios reales que hay en GitHub. Hemos buscado los repositorios más famosos y grandes de Python en dicho portal para formar nuestra base de análisis.

Pensando en la variedad, para obtener datos de un mayor número de desarrolladores, hemos querido coger el mayor número de repositorios posibles, en vez de un solo repositorio muy grande, pensando que quizás en un mismo proyecto se puede trabajar bajo unas mismas pautas marcadas de antemano. Por tanto, aunque dicho proyecto cuente con varios desarrolladores, todos trabajarían bajo las mismas normas y nuestro estudio quedaría reducido a ese único proyecto.

La conclusión que nosotros queremos sacar en este proyecto sobre el problema explicado es a nivel global, y aunque el volumen analizado no sea todo lo extenso que se quiere, ya que sabemos que a este nivel no podemos analizar todo GitHub por así decirlo, sí que hemos querido que los datos tengan la variedad para que así sea.

El estudio más amplio que hemos realizado hasta el momento tiene los siguientes datos:

- **Número total de líneas analizadas:** 369.911

De estas líneas, y con la nomenclatura explicada en el apartado 4, hemos encontrado:

- **Número de líneas verdes:** 330.900
- **Número de líneas naranjas:** 20.107
- **Número de líneas rojas:** 18.904

FIXME: aquí vendría bien un diagrama de barras.

Lo primero que nos llama la atención de estos datos es la cantidad de líneas que han salido de cada tipo, es decir, en un principio mi idea era que todo el código que hemos analizado, o casi todo, seguiría la guía de estilo PEP8, por tanto apenas encontraríamos “línea roja”.

Analizando este dato, vemos que aunque el número de líneas (18.904) nos puede parecer grande en un principio, hay que pensar que tan sólo es el 5,11 %, que es un porcentaje bastante bajo. Al no ser este el grupo de líneas que nos interesa no hemos entrado demasiado en detalle, aunque sí que hemos hecho una comprobación en nuestro código. Hemos querido comprobar cuantas de estas líneas pertenecen a comentarios (tanto comentarios *in-line*, como comentarios de varias líneas), para comprobar si este dato está falseado y no son líneas *reales* de código fuente en la zona roja.

Este análisis nos ha dado como resultado 1.000 líneas, es decir, un 5,28 %. Lo que quiere decir que sí hemos encontrado líneas de código fuente que están en nuestra zona roja.

Si nos centramos en los grupos de líneas que tenemos menores a 80 caracteres, podemos ver que la inmensa mayoría están en las llamadas “línea verdes” un 89,45 %. Esto quiere decir que el grupo conflictivo que nosotros queremos estudiar, “línea naranjas”, queda en un 5,43 %.

La gran mayoría de las líneas están por debajo del umbral que marca la guía de estilo PEP8, incluso, por debajo 70 caracteres, es decir, 10 menos del límite. Por otro lado, tenemos un pequeño porcentaje de líneas que han pasado de los 80 caracteres y que no están cumpliendo la guía de estilo PEP8. Por último, tenemos otro pequeño porcentaje de líneas que están en nuestro rango *conflictivo*, y las cuáles debemos analizar en más profundidad para poder llegar a una conclusión.

## 5.2. Resultados del análisis del grupo conflictivo

En esta sección analizamos con detalle el grupo que hemos considerado *conflictivo*. Después de pasar nuestro primer filtro (4.2), las primeras líneas que podemos considerar *conflictivas* según nuestros parámetros, son las siguientes:

- **Número de líneas conflictivas: 396**

Esto nos da:

- 0.10 % del total de líneas analizadas (369.911)
- 1.96 % del total de líneas en zona naranja (20.107)

Como podemos ver estos datos son unos porcentajes bajísimos, prácticamente ninguna línea se presta a que le pueda ocurrir el problema planteado.

De hecho, todavía queda analizar esas 396 líneas, las cuales han pasado el filtro para ser conflictivas, pero todavía hay que clasificarlas según su motivo, de la forma que ya explicamos en el capítulo 4, (4.5)

- **La línea se acorta exclusivamente por tabulaciones: 1**
- **La línea se acorta exclusivamente por que una palabra pasa a ser mas corta: 125**
- **La línea se acorta por tabulaciones y aparte porque una sola palabra pasa a ser más corta: 3**

- **Resto de líneas:** 267

Como ya vimos, tenemos dos grupos que pueden entrar dentro de nuestro estudio, que son: “La línea se acorta exclusivamente por que una palabra pasa a ser mas corta” y “La línea se acorta por tabulaciones y aparte porque una sola palabra pasa a ser más corta”. Si los agrupamos y hablamos de porcentajes en función de las 396 líneas que ya hemos filtrado como conflictivas, obtenemos:

- **Grupo conflictivo:** 32,32 %
- **Grupo NO conflictivo:** 67,68 %

Como podemos observar en todos los datos mostrados hasta ahora, el porcentaje de líneas que entran en nuestro estudio es mucho menor que el porcentaje que se descarta.

Si comparamos este último filtrado con los datos iniciales, tenemos que **128 líneas de 369.911** pueden haber sufrido el cambio que explicamos en este proyecto a propósito. Lo cual es un **0.034 %**.

Este dato es un porcentaje ínfimo y despreciable en cualquier caso.

Por último, cabe destacar que las 109 líneas que hemos obtenido, puede que no hayan sufrido el cambio a propósito, es decir, esa afirmación es totalmente subjetiva, quedaría el ejercicio de que un humano revisase esas líneas y decidiese si el motivo por el que la línea ha pasado de tener más de 80 caracteres a tener menos de 80 acortando una sola palabra ha sido el problema que se plantea en este proyecto.

FIXME Para ilustrar esta parte, vamos a coger una pequeña muestra de nuestros resultados, y vamos a analizarlos aquí:

(Anulamos las tabulaciones iniciales de todas ellas)

**Par de líneas:**

```
Línea Actual ----->      _VALID_URL =
r'https?:/(?:www\.)?alpha\.com/videos/(?P<id>[^\s/]+)'
```

```
Línea en el Pasado ---->      _VALID_URL =
r'https?:/(?:www\.)?alpha\.com/videos/(?P<display_id>[^\s/]+)'
```



**Opinión:**

En este caso vemos que la palabra que se acorta no parece mucho una variable que haya creado el desarrollador. No tenemos más contexto del resto del código que tiene esta línea alrededor, pero por el nombre de la variable inicial y por el contenido de esta, parece que la parte que se acorta no es una variable en sí. Se puede pensar que depende del programa, el desarrollador esté creando la url él y decida su contenido, pensando así que lo ha podido acortar por nuestro motivo, sin embargo, personalmente no me parece el caso, normalmente trabajas con url que ya existen, y no parece ser que el desarrollador haya acortado una variable suya por el motivo que tratamos. Por tanto yo diría NO.

**Par de líneas:**

```
Línea Actual ----->          return
self.playlist_result(entries, video_id, title, description)

Línea en el Pasado ---->          return
self.playlist_result(entries, video_id, playlist_title, description)
```

**Opinión:**

En esta línea diría nada más verla que SÍ. Podemos observar de manera fácil que lo que ha cambiado es una variable, ya que es un parámetro que se le está pasando a una función. La variable ha pasado de ser “playlist-title” a simplemente “title”. En este caso creo que podemos afirmar que el nombre de la variable ha perdido sentido, ya que llamar a una variable título, no es muy buena elección, antes sabíamos que era el título de una playlist, sin embargo ahora no podemos tener idea del contenido de esa variable, el código ha quedado pero que el anterior, siempre bajo mi punto de vista.

**Par de líneas:**

```
Línea Actual ----->          'skip': 'redirect to
http://swrmediathek.de/index.htm?hinweis=swrlink',

Línea en el Pasado ---->          '_skip': 'redirect to
http://swrmediathek.de/index.htm?hinweis=swrlink',
```

**Opinión:**

En esta línea tenemos un caso en el que parece claro que la palabra que se ha acortado es una variable, sin embargo, no parece que la variable haya perdido sentido respecto a su nombre anterior por este motivo, ya que lo único que se ha hecho ha sido quitar un guión bajo inicial. Por tanto para esta línea también diría que NO.

Lo que sí podemos afirmar es que con los resultados obtenidos en este proyecto somos capaces de llegar a una conclusión.

# Capítulo 6

## Conclusiones

### 6.1. Consecución de objetivos

En el capítulo 2 (2.1) y (2.2) de esta memoria se exponen una serie de objetivos, tanto principales como secundarios, los cuales, después de desarrollar los capítulos (3), (5) y (4), podemos decidir y exponer si se han cumplido o no.

En primer lugar, se planteaba como gran objetivo principal ser capaces de realizar un amplio estudio sobre código Python para poder llegar a una conclusión sobre si los desarrolladores estaban realizando en líneas generales una determinada práctica.

Para ello necesitábamos que nuestro estudio fuese lo suficientemente amplio como para que los parámetros utilizados fueran reales para poder dar una conclusión a nivel global, también necesitábamos que nuestro código tuviese los filtros suficientes, y la capacidad de acotar lo suficiente nuestro problema como para dar justo el resultado que necesitamos.

Creo que este objetivo se ha cumplido completamente, ya que si bien es cierto que el estudio podría ser más amplio, o que en los resultados se podría haber acotado incluso más, los datos obtenidos después de analizar los puntos 4 y 5 de la memoria son suficientes para llegar a una conclusión sobre el estudio propuesto. Dicha conclusión es que los desarrolladores de código Python no están realizando la práctica que aquí se debate, ya que los resultados son claramente negativos.

En este punto también se plantea otro objetivo, quizás más secundario, pero sobre todo más abstracto. Planteamos la idea, sobre todo a nivel de estudiante, de plantearse que es realmente un código de calidad, y la correcta utilización de algunas reglas o guía de estilo como PEP8 o similares.

No se sí se puede decir que esto sea un objetivo en sí, y si lo es, que se haya cumplido como tal. Creo que este proyecto en sí no se ha centrado en expandir esta idea, quizás si algún alumno lee esta memoria sí que pueda obtenerla. Sin embargo puedo decir que este objetivo se ha cumplido con creces en mi persona. Antes de realizar este proyecto no me había planteado tanto la diferencia entre un código que funciona y un código que es bueno.

En cuanto a los objetivos secundarios que comentamos también en el capítulo 2 (2.2):

- **Trabajar de una forma orientada al mundo profesional:** Este objetivo se ha cumplido con el uso de GitHub, centrándonos para el desarrollo de código en GitFlow. También lo he notado en que es un trabajo en el que estás mucho más solo que en el resto de tu vida universitaria, esto te permite enfrentarte sólo a los problemas, organizarte y dividir tu proyecto de la forma que creas necesaria.
- **Aplicar tecnologías e ideas no vistas durante la vida universitaria:** Creo que este objetivo también se ha cumplido. En este proyecto he buscado en todo momento utilizar tecnologías que no he visto en mi vida universitaria o aplicarlas de distinta forma. Por eso en el caso de hacer un desarrollo en la parte del servidor en una posible futura aplicación cliente-servidor he utilizado Node.js y JavaScript como lenguaje. En la universidad hemos visto JavaScript, pero únicamente en la parte del cliente, que es lo más común. En la universidad sólo habíamos trabajado en servidor con java y Python con Django. Para el tema de las bases de datos solo hemos visto SQL, trabajando sólo con bases de datos relaciones, por ese motivo decidí utilizar Mongo, para trabajar con una base de datos no relacional y poder captar las diferencias entre una y otra a la hora de utilizar realmente las dos.

Otra cosa que me gustaría destacar aquí es la elección de la estructura final del proyecto, ojeando proyectos anteriores como hacemos todos antes de realizar el nuestro he visto que casi todos se plantean como una aplicación cliente-servidor con Python en el servidor,

JavaScript en el *front-end*, y SQL como base de datos. He querido enfocar este proyecto como algo un poco distinto, y en mi caso, aunque es cierto que he querido crear una estructura cliente-servidor, para aprender más sobre el tema y utilizar *frameworks* nuevos como Express, el desarrollo se ha centrado más en un estudio en el que queremos obtener unos resultados. El proyecto no ha tenido parte *front-end* como tal.

## 6.2. Aplicación de lo aprendido

Durante la realización de mi grado en la universidad Rey Juan Carlos me he formado en diferentes áreas y campos, muchos de ellos me han facilitado la realización de este proyecto. Algunas de ellas han sido:

1. La programación: La base de este proyecto es la programación, y antes de entrar en la universidad, prácticamente yo no sabía lo que era eso (cabe destacar que ahora que he terminado, me dedico y quiero dedicarme toda la vida a ello). Algunas asignaturas como Fundamentos de la programación, Programación de Sistemas de Telecomunicación o Sistemas Operativos me han enseñado lo que es la programación a bajo nivel.
2. Una vez que sabemos (o algo parecido) lo que es el paradigma de la programación a bajo nivel, hablando a más alto nivel, he aprendido todo lo relacionado con la programación en la parte del servidor en asignaturas como Servicios y Aplicaciones Telemáticas o Ingeniería de Sistemas de Telecomunicaciones.
3. El aprendizaje de GitHub ha sido para mi muy importante para este proyecto. Es otra de las cosas que puedo decir que no sabía antes de empezar la carrera, ni GitHub ni cualquier control de versiones, esto es algo que me enseñó la asignatura Ingeniería de Sistemas de la Información. Además, puedo decir que sé a ciencia cierta que será algo que utilice constantemente en mi vida laboral.
4. Por último, y para nada menos importante, está el concepto general de ser capaz de afrontar cualquier problema sin miedo y ser capaz de resolverlo, en este caso concreto el enfrentarme a nuevas tecnologías que nunca he utilizado y no tener miedo a utilizarlas. Esto es algo que no se aprende en ninguna asignatura en concreto pero que sin duda es algo con lo que salimos de nuestra vida estudiantil.

### 6.3. Lecciones aprendidas

Como ya he comentado en alguna parte de la memoria, uno de mis objetivos era buscar tecnologías y formas de trabajar que no hubiera visto en la universidad, esto al principio fue bastante costoso, sin embargo creo que en el futuro puede tener bastantes beneficios, algunas de las cosas para las que esto me ha servido han sido:

1. Montar un servidor con NodeJS implementado en JavaScript utilizando un framework desconocido para mi como Express.
2. Utilización de una base de datos desconocida para mi como es MongoDB, y su conexión con el servidor.
3. Enfocar un proyecto de grandes dimensiones desde el principio, y planificarlo de forma correcta uno mismo, tanto temporal como estructuralmente.
4. El uso de  $\text{\LaTeX}$ . Una tecnología con la que he escrito esta memoria, y que aunque al principio parece muy farragosa, enseguida ves la facilidad que te aporta para redactar documentos de este tipo.

### 6.4. Trabajos futuros

Todo software implementado tiene margen de mejora, en este proyecto tenemos varios puntos que merece la pena destacar.

1. En primer lugar, y como ya hemos comentado también en algún punto, una mejora muy clara es montar la parte *front-end* y hacer de este estudio una aplicación en la que quizás se puedan mostrar los resultados de forma más atractiva al usuario, y quizás añadir alguna funcionalidad o añadir algún otro estudio para dar más contenido a la aplicación.
2. Mejorar la rapidez de la aplicación para poder analizar un mayor volumen de código. La base del análisis esta programada con varios hilos en paralelo para cada repositorio, esto hace que si el repositorio es demasiado grande y la capacidad del ordenador no es demasiado buena, se creen muchos hilos y pueda dar problemas de rendimiento.

3. Mejorar el filtrado para quedarnos con las líneas determinadas. En este proyecto creo que hemos conseguido afinar bastante, pero aun así, siempre se cuele la llamada basura, y en los resultados finales hemos tenido que dejar un apartado de “Resto de líneas”.
4. Mejorar el análisis de este mismo problema pensando otras opciones. Por ejemplo, hemos tenido en cuenta que el desarrollador cambia para ello una sola variable, o una sola variable y tabulaciones, pero ¿y si acorta dos variables en una misma línea? Ese caso no lo tenemos contemplado.

## 6.5. Valoración personal

Creo que en este punto es hora de echar la vista atrás y reflexionar. Echar la vista atrás al principio de este proyecto y ver como ha quedado finalmente, de lo cual estoy, sinceramente orgulloso. Los resultados obtenidos como proyecto en sí me parecen más bien normales, de lo que sí me siento orgulloso es de como lo he tenido que compaginar con un trabajo a jornada completa desde su inicio, y aunque he tenido que hacer parones no deseados en su elaboración y no he tardado lo que me gustaría, lo he conseguido. También me gusta que es un proyecto algo distinto, en mi opinión, a muchos de los últimos que he visto. Creo que también es hora de echar la vista más atrás y reflexionar sobre la vida universitaria, con este proyecto termino una etapa de mi vida, una etapa muy importante en la que he adquirido conocimientos, habilidades, he perdido miedos, me he hecho una persona adulta y todo ello me servirá para el resto de mi vida, tanto profesional como personal. Cierro esta etapa con mucha alegría y muy orgulloso, y empiezo una nueva con mucha ilusión.





# Apéndice A

## Instalaciones necesarias

Todo lo instalado se ha hecho sobre Windows 10.

### A.1. NodeJS

Lo primero que necesitamos instalar en este caso es Node en nuestro ordenador. Al igual que muchas aplicaciones, Node tiene su propio MSI(Microsoft Installer), el cual podemos descargar desde la página principal de Node e instalar fácilmente en nuestro ordenador siguiendo todos los pasos.



Figura A.1: Descarga de NodeJS

## A.2. NPM

Node Package Manager es un gestor de paquetes que nos ayuda a gestionar las dependencias de nuestro proyecto.

Entre otras cosas nos permite instalar librerías o programas de terceros, eliminarlas o mantenerlas actualizadas.

Generalmente se instala conjuntamente con Node.js de forma automática.

NPM se apoya en un fichero llamado `package.json` para guardar el estado de las librerías. Lanzamos el siguiente comando para realizar la creación del `package.json` e iniciar su configuración.

```
npm init
```

### ■ NPM - package.json

```
{
  "name": "myapp",
  "version": "1.0.0",
  "description": "Esta es la descripción",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Javier Miguel",
  "license": "ISC"
}
```

Figura A.2: Formato del `package.json` después de lanzar el comando `npm init`

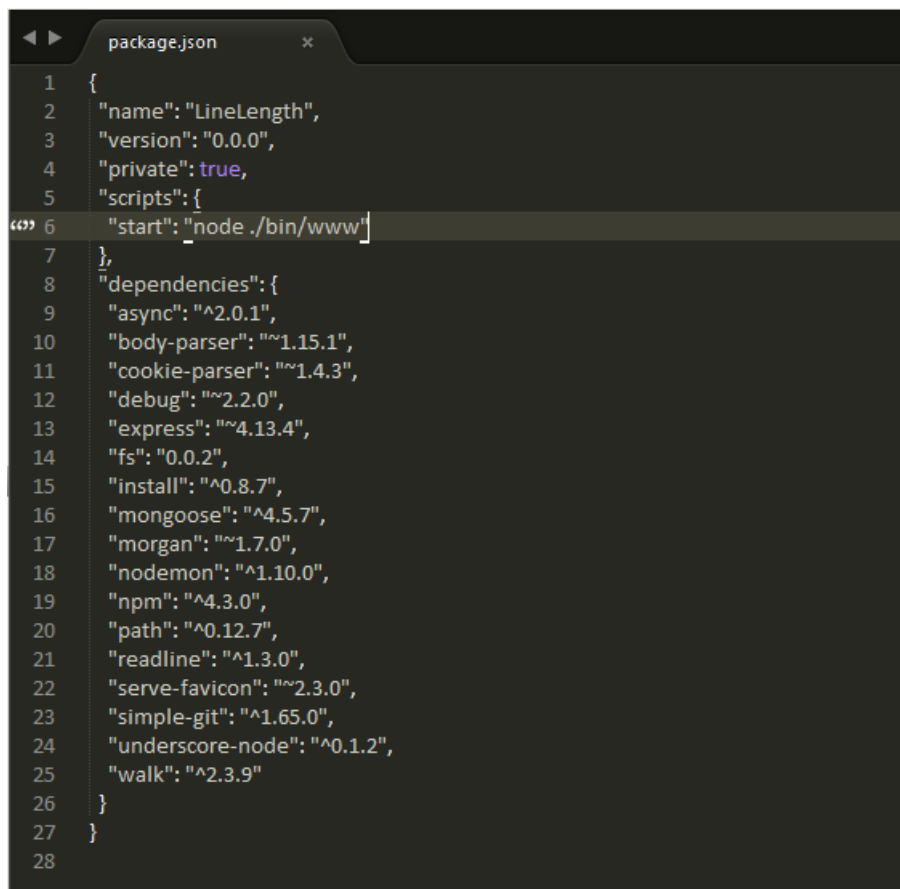
Una vez tenemos el `package.json`, cada vez que queramos incluir una librería nueva a nuestro proyecto, debemos lanzar el siguiente comando:

Por ejemplo, en este caso, si queremos incluir la librería de Express, explicada en el apartado 3, lanzaríamos el siguiente comando.

```
npm install express --save
```

A medida que vayamos necesitando incluir más librerías a nuestro proyecto, iremos ejecutando este comando con cada una de ellas.

Cuando tengamos todas las librerías añadidas a nuestro proyecto, nuestro `package.json` tendrá un formato parecido a este:



```
1 {
2   "name": "LineLength",
3   "version": "0.0.0",
4   "private": true,
5   "scripts": {
6     "start": "node ./bin/www"
7   },
8   "dependencies": {
9     "async": "^2.0.1",
10    "body-parser": "~1.15.1",
11    "cookie-parser": "~1.4.3",
12    "debug": "~2.2.0",
13    "express": "~4.13.4",
14    "fs": "0.0.2",
15    "install": "^0.8.7",
16    "mongoose": "^4.5.7",
17    "morgan": "~1.7.0",
18    "nodemon": "^1.10.0",
19    "npm": "^4.3.0",
20    "path": "^0.12.7",
21    "readline": "^1.3.0",
22    "serve-favicon": "~2.3.0",
23    "simple-git": "^1.65.0",
24    "underscore-node": "^0.1.2",
25    "walk": "^2.3.9"
26  }
27 }
28
```

Figura A.3: Formato nuestro `package.json`

### A.3. Creación de un nuevo proyecto Express

Una vez tenemos el equipo listo, pasamos a crear un nuevo proyecto. En el punto 3, hemos explicado en que consiste Express, el framework de Node.js con el que vamos a realizar nuestro proyecto.

Dentro de Express vamos a utilizar la librería de Express Generator, la cual nos crea una estructura base para una aplicación.

Para ello ejecutamos en el directorio en el que queramos crear nuestro proyecto los siguientes comandos:

Primero instalamos express-generator de forma global, añadiéndole el -g

```
npm install express-generator -g
```

Seguidamente lanzamos:

```
express <nombreApp> [--ejs]
```

```
cd <nombreApp>
```

```
npm install
```

Con esto se nos creará un proyecto con la siguiente forma:

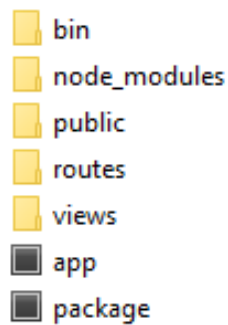


Figura A.4: Estructura básica de un proyecto

Dentro de la carpeta *routes* meteremos el código JavaScript que ejecutaremos en la parte del servidor, y que tendrá toda la lógica de nuestro proceso.

# Bibliografía

- [1] Node.js. <http://www.netconsulting.es/blog/nodejs/>. Accessed: 2016-03-15.
- [2] Express. <https://geekytheory.com/introduccion-a-express-js>.
- [3] Express-2. <https://pixelovers.com/ventajas-utilizar-nodejs-1953900/>.
- [4] Express-3. <https://github.com/senchalabs/connect#readme>.
- [5] Express-4. <http://www.sinatrarb.com/documentation.html>.
- [6] MongoDB. <https://www.mongodb.com/es>.
- [7] JSON. <https://geekytheory.com/json-i-que-es-y-para-que-sirve-json/>. Accessed: 2016-04-10.
- [8] GIT. <https://git-scm.com/book/es/v1/Empezando-Fundamentos-de-Git>.
- [9] GITFLOW. <https://sysvar.net/es/entendiendo-git-flow/>.
- [10] ESPRESS-GENERATOR. <http://malnuer.es/js/crear-una-app-nodejs-con-express-generator/>.
- [11] PEP8. <https://alexanderae.com/pep8-guia-de-estilo-para-python.html#fn-1>.

