



MÁSTER DE INGENIERÍA DE TELECOMUNICACIÓN

Curso Académico 2018/2019

Trabajo Fin de Máster

ANÁLISIS Y ESTUDIO DEL USO DE *PYTHONIC*
IDIOMS

Autor : José Javier Merchante

Tutor : Dr. Gregorio Robles

Trabajo Fin de Máster

Análisis y Estudio del Uso de *Pythonic Idioms*

Autor : José Javier Merchante

Tutor : Dr. Gregorio Robles Martínez

La defensa del presente Proyecto Fin de Carrera se realizó el día de
de 2019, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2019

*Dedicado a
mis padres y mi hermana*

Agradecimientos

La realización de este trabajo de fin de máster supone el final de una gran etapa de mi vida. Se queda grabados recuerdos de estrés, de aprobados ajustados y matrículas de honor tras tanto tiempo dedicado a cada una de las asignaturas. Los momentos en los que la compaginación de trabajo y estudios no eran tan sencillos como la gente afirmaba.

Todo este esfuerzo, por fortuna, no lo he realizado solo, sino que la compañía siempre ha sido una clara aliada, con un apoyo constante que al final ha dado sus frutos.

Es por ello que quiero agradecer a mi familia, por esos momentos en los que estuvieron preocupándose por mí, animándome a continuar cuando las circunstancias no eran los mejores, y alegrándose por cada una de las metas que juntos hemos conseguido.

También agradecer a mis compañeros de clase por todo este tiempo compartido juntos, estando unidos como una pequeña familia, tanto para lo bueno como para lo malo, y con ese “pique” constante por buscar siempre la mejor nota de la clase.

Por último, y no menos importante, agradecer a Gregorio estos años que tanta experiencia que me ha aportado. Tanto en el pasado de manera curricular, como trabajando junto a él, así como en lo personal. Siempre ha creído y confiado en mí, y ha buscado continuamente un “JJ” que estuviera en lo más alto, gracias.

A todos ellos les debo este paso en mi vida, ¡gracias!

Resumen

En este proyecto se realiza un estudio acerca de los elementos avanzados del lenguaje de programación Python.

Para el desarrollo del mismo, se han realizado entrevistas a programadores para conocer su percepción acerca del significado y aplicación de estas estructuras *pythonic*. A partir de los resultados, se han podido obtener unas conclusiones precisas que podrán ser de utilidad para estudios relacionados y que demuestra la importancia del proyecto que se está realizando.

Por otro lado, se ha realizado la implementación para el análisis de repositorios escritos en Python. Cualquier programador, tanto experto como principiante, puede realizar un análisis de sus repositorios mediante una herramienta web, y obtener unos resultados acerca de los mismos para mejorar en el ámbito de Python.

Este proyecto se enmarca en un trabajo de investigación para la identificación y conocimiento de la cultura de uno de los lenguajes de programación más usados hoy en día. Se busca mejorar la comprensión del término *pythonic* así como incentivar el uso de dichas estructuras.

Summary

This project is a study about *Pythonic idioms*.

Interviews to programmers have been conducted in order to gather information about the knowledge and application of *pythonic idioms*. Based on the results, it has been possible to obtain some conclusions that can be used for related studies, and shows the importance of the project that is being carried out.

On the other hand, we have made a implementation for the analysis of Python repositories. Any programmer, both expert and beginner, can perform an analysis of their repositories through a web tool and obtain some results about them to improve in the field of Python.

This project is part of a research work of one of the most important languages used today. The aim is to improve the understanding of the term *pythonic*, as well as to encourage the use of these structures.

Índice general

1. Introducción	1
1.1. Conceptos previos	2
1.2. Motivación	2
1.3. Contexto	3
1.4. Estructura de la memoria	5
2. Objetivos	7
2.1. Objetivo general	7
2.2. Objetivos específicos	8
3. Estado del arte	9
3.1. Python	9
3.2. <i>Django</i>	10
4. Análisis cualitativo de elementos <i>pythonic</i>	13
4.1. Características las entrevistas realizadas	14
4.2. Preguntas realizadas	15
4.3. Resultados de las entrevistas	16
4.3.1. Significado del término <i>pythonic</i>	16
4.3.2. Cómo se aprende y se aplica el código <i>pythonic</i>	18
4.3.3. Impacto y propagación de la cultura en el trabajo y en la comunidad	20
5. Aplicación web de análisis de proyectos Python	21
5.1. Introducción a la plataforma	21
5.2. Tecnologías utilizadas	22

5.3. Esquema de análisis de repositorios	23
5.4. Recopilación de <i>idioms</i> con ejemplos	27
6. Conclusiones	31
6.1. Consecución de objetivos	31
6.2. Lecciones aprendidas	32
6.3. Trabajos futuros	33
A. Preguntas realizadas en las entrevistas	35
Bibliografía	37

Índice de figuras

5.1. Gráfico de petición de análisis de un usuario	24
5.2. Página principal	25
5.3. Selección de repositorios	25
5.4. Resultados de un usuario	26
5.5. Resultados de un repositorio. Indica la localización de cada <i>idiom</i>	27
5.6. Explicación de <i>list comprehension</i> en la web	29

Capítulo 1

Introducción

En la era en la que vivimos, en múltiples disciplinas y sobre todo enfocados en el ámbito de la telecomunicación, el manejo de la información es un pilar clave. El uso de la programación para el análisis de los datos es imprescindible para poder filtrar y generar nueva información que podría ser de más utilidad.

En la actualidad, la cantidad de código escrito por programadores de diferentes ámbitos aumenta considerablemente. Desde crear nuevos programas que faciliten la realización de determinadas tareas de cualquier trabajador, hasta herramientas enfocadas en el análisis de datos para conocer la evolución de un producto.

La importancia de mantener el código “limpio”, es decir, siguiendo unos determinados patrones de diseño, es bastante determinante para poder ser mantenido para solucionar futuros errores o incluso implementar nuevas características.

La evolución de los proyectos almacenados en plataformas de desarrollo colaborativo, como puede ser el caso de GitHub o GitLab entre otros, aumenta significativamente, tanto en número, como en el contenido de los mismos.

La relevancia del *Open Source* (código que está disponible públicamente bajo unas condiciones determinadas que permiten copiar, modificar y redistribuir el software) ha ido cobrando mayor importancia a lo largo de los últimos años. Las grandes compañías cada vez optan más por publicar su software, aunque sólo sea de algunas aplicaciones, como en el caso de *Microsoft*.

La publicación de código en abierto supone un esfuerzo adicional por parte de las organizaciones que lo mantienen. El código siempre debe mantenerse *limpio*, es decir, seguir unos determinados patrones de diseño para poder ser mantenido para futuros errores, pero además, si

las compañías quieren beneficiarse de uno de los beneficios del código abierto, como es el caso de contribución externa al proyecto, es importante reflejar una buena imagen del código.

Es por cuestiones como las mencionadas, que se hace necesario el uso de herramientas que puedan ayudar o facilitar este tipo de tareas. En el presente proyecto vamos a poner el foco en uno de los lenguajes más populares en los últimos años, Python. En el momento de escribir esta memoria, se encuentra el tercero en el índice TIOBE¹, y es uno de los lenguajes que más ha crecido en los últimos años.

Este lenguaje tiene a su disposición un gran número de herramientas como es el caso de *pep8* o *pylint* que ayudan a seguir una serie de convenios en la sintaxis. Sin embargo, nosotros queremos hacer un estudio que vaya más allá de las reglas del estilo y enfocarlo más a las estructuras utilizadas.

El uso de determinadas elementos denominados *idioms*, facilitan en gran medida la programación en un lenguaje, y por lo tanto, el aprendizaje y aplicación de los mismos favorecen a un código más limpio, más sencillo de leer, e incluso en algunos casos, mejora el rendimiento de la ejecución al ser estructuras más optimizadas.

1.1. Conceptos previos

1.2. Motivación

En la actualidad, tal y como se comentó anteriormente, se hace necesario un estudio para mejorar el estado del código de muchos programadores. Existen cantidad de ellos que por exigencias personales o incluso por necesidades de las empresa necesitan enriquecer su código.

Para ilustrar un ejemplo sencillo, un programador principiante en Python, y que quizá tiene más experiencia en otros lenguajes, podría programar el siguiente extracto de código:

```
positions = ['Ana', 'Juan', 'Pepe', 'Marta']
i = 0
for name in positions:
    i += 1
    print(i, name)
```

¹<https://www.tiobe.com/tiobe-index/>

Se puede observar una iteración sobre una lista, y mediante una variable auxiliar, imprime por pantalla las posiciones en las que han quedado unos corredores. Este código ejecuta a la perfección, hace su labor y sigue las reglas de estilo en Python. Sin embargo, existe una manera más “bonita” y eficiente de hacerlo, que es lo que sería considerada por la comunidad un código más *pythonic*:

```
positions = ['Ana', 'Juan', 'Pepe', 'Marta']
for i, name in enumerate(positions, 1):
    print(i, name)
```

Tal y como puede observarse, el número de líneas ahora es menor, es más sencillo de leer, y además, se puede comprobar que con un bucle de una lista de mayores dimensiones se mejora el tiempo de ejecución del programa.

Por lo tanto, la necesidad de aplicaciones que nos permitan conocer este tipo de estructuras o identificar cambios en nuestro código puede ser de gran ayuda tanto a programadores principiantes como a aquellos que quieran conocer más acerca del lenguaje.

Recurrir a varios libros y páginas webs para leer con todo este contenido siempre es una opción válida, pero conocer nuestros errores mientras programamos o conocer nuevas estructuras agrupadas en una única web puede ser siempre una manera más útil de aprender.

1.3. Contexto

Como contexto de este Trabajo de Fin de Máster se encuentra el desarrollo de una herramienta de búsqueda de *idioms*, la publicación de pequeños artículos en seminarios internacionales, participación en la PyConES como ponente y la colaboración con la universidad de Zúrich para la publicación un artículo en una conferencia.

El programa para la identificación de *pythonic idioms* se desarrolló para un Trabajo de Fin de Grado. El autor y el director fueron los mismos que para el presente proyecto. Para la realización de la aplicación se generó una lista de en torno a 100 *idioms* documentándose a partir de diferentes libros, sitios web, foros, e incluso por experiencias personales. La herramienta consistía en un *script* de Python que seguía el siguiente guión:

- El usuario introducía en la llamada un repositorio *git*.

- El programa se encargaba de clonar el repositorio en local para obtener el código.
- Posteriormente se identificaban y seleccionaban los ficheros que fueran Python, es decir, terminados en *.py*, o sin extensión y la primera línea (*shebang*) indique que es un fichero Python.
- Por cada uno de los ficheros, se dividía el código en *tokens* con el *lexer* de la librería *Pygments* (un resaltador de sintaxis).
- Se recorría la cadena de *tokens* y se identificaban cada uno de los *idioms*. En unos casos eran simples al ser llamadas a funciones, y en otros casos se buscaban determinados patrones.
- Por cada uno de los *idioms* encontrados se almacenaba la línea y fichero donde se habían encontrado y además se intentaba identificar el autor del mismo. Para ello se miraba con *git blame* el correo de la última persona que había modificado la línea.
- Finalmente se ofrecían al usuario todos los *idioms* identificados.

Este programa ha servido de base para la realización de la aplicación web con alguna modificación para adaptarlo a la recogida de los datos.

Por otro lado, es importante señalar la colaboración con profesores y un estudiante de doctorado de la Universidad de Zúrich. Junto a ellos, se elaboró una publicación de un artículo en una conferencia. En dicho artículo, se realiza un análisis de cómo los programadores entienden el término *pythonic* a partir de entrevistas, se muestra un catálogo de *idioms* a partir de la literatura y se exponen los efectos de tener en un lenguaje un término específico para la calidad del código, considerando el potencial que podría tener para otros lenguajes y ecosistemas [1].

Cabe mencionar en el contexto de este trabajo, la participación en pequeños seminarios internacionales de software enfocados a presentar el trabajo en progreso. Estos seminarios fueron SATToSE² y BENEVOL³. Se enfocaban principalmente en presentaciones o artículos relacionados con las técnicas y herramientas para la evolución del código.

Finalmente destacar que se realizó la presentación de la herramienta web para el análisis de repositorios en la PyConES, la conferencia Python más grande de ámbito nacional, a partir de

²<http://sattose.org/>

³<http://ansymore.uantwerpen.be/benevol-2017-agenda>, <https://benevol2016.wordpress.com/program/>

la cual se obtuvieron candidatos para la realización de entrevistas acerca del código *pythonic*.

1.4. Estructura de la memoria

En esta sección se detalla la estructura en la que se ordena la presente memoria para facilitar su comprensión:

- En la introducción del capítulo 1 se realiza una descripción del proyecto incluyendo algunos conceptos claves, mostrando la motivación adherida al trabajo y exponiendo el contexto derivado del estudio.
- En el capítulo 2 se encuentran los objetivos, generales y específicos, que se intentan abordar en este trabajo.
- A continuación en el capítulo 3 se mostrarán las tecnologías utilizadas en el proyecto y una breve pero explicativa descripción de las mismas.
- Situado en el capítulo 4 se explica la realización de las entrevistas. Se enumeran las categorías de preguntas que se han realizado, y los resultados obtenidos de las mismas.
- Por otro lado, en el capítulo 5 se realiza una explicación extensa del uso de la aplicación web de análisis de repositorios.
- Tras la explicación de lo anterior, en el capítulo 6 se muestran las conclusiones finales del proyecto, objetivos alcanzados y una breve valoración personal.

Capítulo 2

Objetivos

2.1. Objetivo general

El objetivo principal de esta memoria es conocer la importancia de escribir código limpio en los programas. Dentro de la comunidad de Python se destaca siempre el código *pythonic*, es por ello que hemos querido profundizar y conocer los diferentes aspectos que son de más relevancia.

Se pretende dar respuestas a preguntas como:

- ¿Qué son según la comunidad los elementos *pythonic*?
- ¿Qué importancia tienen en las empresas y en plataformas de software colaborativo?
- ¿Cuál es la mejor manera de aprenderlos?

Por otro lado, se busca conocer el análisis de repositorios de usuarios para mejorar su estudio. Entre los objetivos destacan:

- Realización de una herramienta web para el análisis de repositorios
- Poder realizar el análisis de manera concurrente de varios usuarios
- Mostrar información agrupada de un listado de elementos *pythonic*

2.2. Objetivos específicos

Como objetivos más concretos y profundizando en la elaboración de entrevistas, se pueden destacar los siguientes puntos:

- Conocer los procesos para la elaboración de entrevistas
- Realizar preguntas enfocadas en los objetivos
- Conocer por qué los programadores deciden utilizar determinados *idioms*
- Saber cómo clasificarían los programadores una lista de elementos *pythonic*

En el sentido de análisis destacar los siguientes objetivos específicos:

- Buscar la sencillez para el despliegue con una interfaz ligera
- Facilitar el aprendizaje de nuevos *idioms*
- Perseguir que fuera usado por bastantes usuarios

Capítulo 3

Estado del arte

3.1. Python

Python¹ es un lenguaje de programación de alto nivel, interpretado, orientado a objetos y con una semántica dinámica.

Es un lenguaje relativamente reciente, en 1994 Guido van Rossum publica la versión 1.0 del código enfocándose en crear un lenguaje de *scripting* con sintaxis sencilla. Posteriormente Python ha ido mejorándose con ayuda de la comunidad para lanzar nuevas versiones que han perfeccionado este lenguaje. Sin embargo, no ha sido hasta los últimos 15 años cuando Python empezó a crecer de manera desmesurada.

El tipado de Python es dinámico, lo que quiere decir que no existe una comprobación del tipo de las variables hasta la ejecución. Por otro lado, los datos son fuertemente tipados, es decir, no se puede usar una variable de un determinado tipo como si fuera de otro distinto como ocurre en otros lenguajes como JavaScript.

Debido a que Python es un lenguaje con estructuras de datos de alto nivel, acompañado de una sintaxis sencilla y la omisión del paso de la compilación, es un lenguaje que tiene un ciclo de edición-testeo muy versátil, lo que lo hace muy querido y deseado en el ámbito del software tal y como se refleja en los estudios anuales de StackOverflow².

La librería estándar de Python, así como los paquetes que desarrolla la comunidad, hace de Python un lenguaje de programación que pueda ser utilizado en una amplia variedad de disci-

¹<https://python.org>

²<https://insights.stackoverflow.com/survey/>

plinas, desde educación debido a la sencillez de su sintaxis hasta medicina para reconocimiento de imagen.

En cuanto al ámbito de la telecomunicación, Python es un lenguaje bastante utilizado. La realización de un servicio web en Python puede conseguirse en pocas líneas de código gracias a *frameworks* como *Django* o *Flask*. Por otro lado, para el análisis de grandes cantidades de datos, así como para el aprendizaje máquina, librerías como *pandas* o *scikit-learn* hacen de este lenguaje uno de los más manejados.

Como último punto a destacar de este lenguaje es la filosofía que se mantiene dentro de la comunidad. Tim Petters publicó una guía de principios básicos para programar en este lenguaje que permanece muy popular entre los programadores más expertos. Se hace llamar The Zen of Python [4]. Un extracto de los 20 aforismas son los siguientes:

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

...

3.2. Django

Django es un *framework* desarrollado en Python que permite la realización de un entorno web de manera sencilla y con poco código. Tiene un diseño pragmático que busca la organización por ficheros para modificar los mínimos detalles.

Está diseñado para evitar errores comunes de seguridad. Django está protegido contra la

inyección de SQL, cross-site scripting, tiene verificación de cabeceras, e incluso protección contra peticiones generadas suplantando la identidad de otro usuario (CSRF)

Está desarrollado bajo la frase de “no reinventar la rueda”. Django ofrece todo lo necesario para poder realizar un servicio web. Desde la autenticación y manejo de sesiones a plantillas y filtros para mostrar el contenido al cliente de manera más dinámica.

Por último mencionar que existe una gran cantidad de *plugins* creados por la comunidad. Por ejemplo para realizar tareas en segundo plano o autenticación en sitios web de terceros como GitHub o Google.

Capítulo 4

Análisis cualitativo de elementos *pythonic*

Las entrevistas, históricamente han sido frecuentemente usadas para estudiar el comportamiento humano. El propósito de utilizarlas en estudios empíricos es para obtener información acerca de un tema que no puede ser moderado usando únicamente métodos cuantitativos. Este tipo de encuentros son un método muy útil para obtener datos, pero sin embargo es una actividad que conlleva mucho tiempo.

La parte más importante para la realización de una entrevista es la preparación. Todas las preguntas deben ser muy precisas y centradas en el tema principal. Suele haber un tiempo limitado para realizar preguntas a los entrevistados y en muchas ocasiones una única oportunidad de realizarlas, por lo tanto, una preparación previa es muy importante para obtener la mayor cantidad e información posible.

La segunda parte más importante de la entrevista es la realización. Existe una gran cantidad de información acerca de cómo desarrollarla. Los puntos más importantes son crear un ambiente de confianza, no expresar desacuerdo con lo que dice el entrevistado, prestar atención para evitar realizar preguntas respondidas y expresar las preguntas con claridad entre otras cuestiones [2].

La última parte, no menos importante, tiene que ver con analizar los resultados de la entrevista. Buenas preguntas y respuestas precisas pueden ayudar en gran parte esta labor. Sin embargo, la transcripción de las respuestas, la unificación de las respuestas y la búsqueda de similitudes y diferencias aumenta en muchas ocasiones el tiempo de realización.

Id	Experiencia en Python (años)	Empleo actual
I1	6	Ingeniero de operaciones
I2	16	Consultor de software, profesor de Python
I3	4	Analista jefe de datos
I4	3	Desarrollador en seguridad
I5	11	Investigador
I6	>6	Director de ingeniería
I7	6	Desarrollador de Software
I8	2	Desarrollador de Software
I9	>10	Director técnico
I10	2-3	Estudiante
I11	3	Analista jefe de datos
I12	1	Desarrollador de Software
I13	9	Automatización de infraestructura

Cuadro 4.1: Desarrolladores entrevistados

4.1. Características las entrevistas realizadas

Para la realización de las entrevistas se realizó una búsqueda de candidatos centrada en personas que conocían el lenguaje con anterioridad sin importar el nivel al que lo usaran. Para ello se buscaron participantes una conferencia nacionalmente conocida denominada PyConES¹ y por correo a personas voluntarias.

Finalmente, realizamos entrevistas a trece candidatos que utilizaban Python en múltiples disciplinas y con un conocimiento del mismo lo suficientemente diverso que iba de 1 año a 15 años de experiencia con el mismo. Los candidatos tenían diferentes nacionalidades pero sobre todo española.

En la tabla 4.1 se pueden ver las características de cada uno de los entrevistados. En la primera columna el identificador, a continuación los años de experiencia en el lenguaje Python, y en la última columna el actual empleo de cada uno de ellos.

Los candidatos utilizaban el lenguaje principalmente en el ámbito profesional. Algunos de

¹<http://pycon.org>

ellos publicaban todo el código que desarrollaban, mientras que otros lo mantenían privado por política de la empresa.

Parte de las entrevistas fueron desarrolladas en persona, mientras que otras debido a la distancia o a los horarios que disponían, tuvieron que realizarse de manera remota mediante software de videollamada.

Todas las entrevistas fueron grabadas en audio para poder recoger la mayor información disponible. La idea inicial era realizar entrevistas de 10 minutos, pero finalmente la duración media fue de 15 minutos por entrevistado debido a que se realizaron más preguntas en función de las respuestas recogidas.

Al principio de cada entrevista, a los entrevistados se les exponía el contexto del proyecto, explicando el propósito de la investigación y evitando en cualquier caso influir en las respuestas como puede ser el caso de la importancia de los elementos *pythonic*.

Una vez terminaron las entrevistas, las grabaciones fueron transcritas a texto, y agrupadas por preguntas genéricas. Para el análisis, los candidatos se clasificaron por diferentes criterios (su experiencia en el lenguaje, el tipo de puesto de trabajo o si las empresas mantenían el código público o privado) para poder encontrar similitudes y diferencias.

Todo el contenido de las entrevistas puede encontrarse en el repositorio en el que se encuentra almacenada esta memoria debido a que la extensión es excesiva para un anexo en esta memoria (Anexo A).

4.2. Preguntas realizadas

Para las entrevistas realizadas, las preguntas estuvieron enfocadas en aquellos aspectos que no pueden ser respondidos fácilmente con análisis de código. Conocer el significado del término *pythonic*, las decisiones tomadas para el uso de elementos culturales o la importancia de su uso en el ámbito laboral son algunos temas importantes de conocer.

Se realizó un conjunto mínimo de preguntas para ser respondidas por los participantes. Las preguntas se ejecutaron de manera diferente dependiendo del desarrollo de la entrevista y de la visión que tenía acerca de los elementos *pythonic*.

Las preguntas de las entrevistas se redactaron de manera semiestructurada [3], la mayoría eran necesarias para todos los participantes, algunas se ejecutaron en orden diferente dependien-

do del desarrollo de la entrevista y de la visión que tenía acerca de los elementos *pythonic*, y otras se realizaron con final abierto que pudieran desembocar en cuestiones con nuevos detalles.

Las preguntas redactadas fueron clasificadas en las tres categorías objeto de estudio:

- Conocer qué es lo que saben y piensan acerca del término *pythonic*.
- Averiguar cómo los entrevistados aprenden y aplican dichas estructuras.
- Comprender el impacto y propagación de esta cultura en el trabajo y en la comunidad.

Todas las preguntas realizadas se encuentran en el Anexo A. El orden pudo variar entre entrevistados dependiendo de las repuestas dadas o el nivel de destreza que pueda tener en el lenguaje.

4.3. Resultados de las entrevistas

Para ilustrar los resultados de las entrevistas, el análisis se va a clasificar en función de la separación previamente mencionada sobre las preguntas.

4.3.1. Significado del término *pythonic*

Esta palabra es ampliamente conocida en el ámbito de Python, sin embargo, el significado de la misma podría variar dependiendo de la disciplina en la que trabajen o los años de experiencia.

Todos los entrevistados conocían o habían escuchado hablar acerca de dicho término. Los participantes definieron *pythonic* como la manera legible y elegante de programar en Python utilizando las estructuras ofrecidas por el lenguaje así como las librerías de las que dispone. Todos ellos mencionaban que mejoraba la lectura del código, y algunos de ellos que mejoraba el rendimiento del programa al ser estructuras optimizadas.

La estructura *pythonic* más mencionada fueron las *list comprehensions* (10 veces). Otros ejemplos aportados fueron las *keywords* (palabras reservadas del lenguaje), reglas de estilo de Python (PEP 8) [5], el uso de estructuras como *for ... in ...* y algunos *built-in* como *enumerate*.

Dependiendo de la experiencia de los programadores, nos encontramos en ligeras diferencias en su concepción acerca del término. Aquellos con más experiencia en el lenguaje, mencionaban los términos *built-in*, eficiencia y estructura con mayor frecuencia. En cambio, aquellos

con menor destreza, se referían más a código con mejor estilo, y a menor líneas de código para resolver una misma tarea.

En relación a otros lenguajes, los entrevistados comentaron que existen también *idioms* o una mejor manera de implementar algunas estructuras, sin embargo, no existe un concepto tan extendido como en Python:

[I2] En Python existe una mayor tendencia a valorarlo. En otros lenguajes es una característica que no persigue la comunidad. Pero Python, desde sus orígenes, quizá por casualidad o por su creación, es mucho más valorado. Es un lenguaje que tiene muchas posibilidades y mucha funcionalidad avanzada que hace más *idioms*. Es algo que viene del lenguaje y en Python es más común que en otros lenguajes.

Otra respuesta define Python como un lenguaje centrado en la legibilidad, que incluye recursos para emprenderlo y que en muchas ocasiones conlleva a escribir menos líneas:

(Inglés)[I5] *Pythonic* is more like using the right tool at the right place, and by “right tool”, I mean everything that’s provided by the language and its standard library.

(Traducción)[I5] *Pythonic* es más como usar la herramienta correcta en el lugar correcto y con la “herramienta correcta”, me refiero a todo lo que proporciona el lenguaje y su biblioteca estándar.

Una respuesta ofrecida por uno de los entrevistado, indicaba que escribir código con muchos *idioms*, no siempre significaba que el código fuera *Pythonic*, tan solo es recurso para mejorarlo:

(Inglés)[I5] While there are many idioms in Python, using them does not mean that you’re writing pythonic code. Sometimes, idioms make the code less readable, or more complicated. For instance, using *reduce* can be seen as idiomatic, but in general results in code that is not easy to understand, and thus is not pythonic to my eyes.

(Traducción)[I5] Si bien hay muchos *idioms* en Python, usarlos no significa que estés escribiendo código *pythonic*. A veces, los *idioms* hacen que el código sea menos legible o más complicado. Por ejemplo, el uso de *reduce* puede verse como *idiomatic*, pero en general da como resultado un código difícil de entender, y por lo tanto no *pythonic* para mis ojos.

Los entrevistados con más experiencia, acentúan que en Python existe siempre una manera sencilla de programar una tarea con *idioms* y con estilo. Otros lenguajes en cambio, tienen ciertas convenciones sobre el estilo, pero no en la manera de resolver un determinado problema.

Todos los programadores mostraron su aceptación por el código *pythonic*, ayuda a mantener el código *limpio*, detectar errores y cometer menos fallos.

4.3.2. Cómo se aprende y se aplica el código *pythonic*

Actualmente, se puede obtener una gran cantidad de información acerca de qué es el código *pythonic* y numerosos ejemplos, sin embargo, el término prácticamente no aparece en cursos o libros del lenguaje. Por ejemplo en libros importantes de introducción al lenguaje como *Learning Python - Mark Lutz*, el término Pythonic aparece en 6 ocasiones (en más de 1200 páginas), en el libro *Python Cookbook - David Beazley & Brian K Jones* se realiza una única referencia, y en *Head First Python - Paul Barry* no se realiza ni una referencia. Por lo tanto, es importante conocer cómo este termino se aprende para perfeccionar nuestro conocimiento y por otro lado cuando se aplica.

La gran mayoría de los entrevistados respondieron que aprendieron acerca del código *pythonic* leyendo código de otros programadores. En la actualidad, una gran parte del código utilizado para la elaboración de programas se encuentran de libre acceso en plataformas como GitHub o GitLab. Algunos de los entrevistados nos ha comentado que entendiendo el código de repositorios populares como es el caso de *requests* (librería para realizar peticiones HTTP más sencillas), aporta nuevo conocimiento y cultura en el lenguaje.

Por otro lado, otros programadores indicaron que aprendieron código *pythonic* a partir de libros, conferencias o compañeros de trabajo. *STACKOVERFLOW* fue mencionado por alguno de los entrevistados para obtener pequeñas recetas, en cambio otros mostraron su desaprobación debido a que no es la mejor fuente para aprender:

[I4] He leído muchos libros de Python y documentación acerca del término Pythonic. Hay momentos en los que te sientes atascado, entonces en *StackOverflow* encuentras diferentes puntos de vista de la gente, y siempre se puede aprender algo.

Según los participantes, convertirse en un programador *pythonic* conlleva mucho tiempo. En relación a esto, alguno de los programadores afirmaban que algunas estructuras o elementos eran mucho más sencillos de entender:

[I10] Al principio yo no las entendía [*pythonic idioms*], pero posteriormente estaba interesado en la documentación y en el mundo que me abría. Me tomó mi tiempo aprender y

tomar partido de los *idioms*

Por otro lado, afirmaban que el código que utilizaba este tipo de estructuras era mucho más sencillo de leer, pero muchas veces no es tan sencillo implementarlo. Uno de los ejemplos que mencionaron como estructura compleja fueron los *decorators* (una función que toma a su vez a otra y altera su comportamiento sin modificar su implementación) y algunas funciones de la librería *itertools* (un módulo que implementa iteradores sobre colecciones).

Los programadores más avanzados afirmaron que su código se había vuelto más *pythonic* con los años, lo que podría reflejar que un código más experimentado es más *pythonic*. Cuando fueron preguntados si serían capaces de diferenciar el código de un programador con experiencia de otro más inexperto, confirmaron que el uso de los elementos *pythonic* pueden ser una buena señal:

[I10] Los programadores *Junior* suelen escribir código simple con algunos errores. Los programadores intermedios, usan una gran variedad de herramientas de la librería de Python, pero a veces el código es complicado de entender. Los más expertos, suelen escribir código simple, legible y con el uso de muchos *idioms*

Algunos entrevistados, nos explicaban que los programadores de Python principiantes se pueden diferenciar porque a veces escriben los nombres de variables o funciones con formato *CamelCase* en lugar de *snake_case*. Además suelen implementar *getters* y *setters*. Cuestiones como las anteriores, pueden confirmar que son programadores que provienen de otros lenguajes y que es muy probable que no sean expertos en Python.

Como aclaración última de un programador experto enfocado a la investigación, nos afirmaba:

(Inglés)[I5] ‘Pythonic’ and Python idioms are different things. [However,] ‘Pythonic’ can be used to measure a developer’s skills, idioms can be used to (at least) measure a developer’s knowledge.

(Español)[I5] *Pythonic* y Python *idioms* son diferentes términos. *Pythonic* puede ser usado para medir la destreza de un programador, los *idioms* pueden ser usadas (por lo menos) para evaluar los conocimientos del programador.

4.3.3. Impacto y propagación de la cultura en el trabajo y en la comunidad

Es importante conocer cuáles son los ámbitos en los cuales este tipo de estructuras tienen un mayor impacto, y conocer en detalle su necesidad en el ámbito laboral.

En la primera pregunta enfocada a la relevancia de su uso en la empresa, nos encontramos que en compañías que creaban código abierto, solía ser necesario mantener un código *pythonic*, ya fuera directamente o mediante técnicas de revisión. Todo *commit* que fuera a un entorno público debería ser revisado para mantener unos estándares de calidad de código. En contraposición, en compañías que no publicaban su código abiertamente, los entrevistados constataron que el código debía estar documentado, pero no necesariamente *pythonic*.

Cuando realizaban entrevistas para un puesto de trabajo, la mayoría de los entrevistados nos afirmaron que escribir estructuras de la manera *pythonic* siempre mostraba un grado más de experiencia, y por lo tanto aumentaban las posibilidades de ser contratado. Uno de los entrevistados [I4] afirmaba que había realizado entrevistas en las que le había mencionado el término *pythonic*, y al ser poco experimentado en el lenguaje, tuvo que documentarse y aprender acerca de la materia para las consecuentes entrevistas.

Los candidatos sostienen que comentar el código con sus compañeros de trabajo favorece a aprender y a aplicar buenas prácticas. Es casi siempre el mejor de los métodos, sin embargo, la falta de tiempo en el trabajo, provoca que no sea una costumbre.

Los entrevistados afirman que modifican su código cuando ven que han escrito algo complejo de entender, pero cuando aprenden una determinada estructura, no suelen volver a mirar dónde la podrían aplicar. Cuando el código se ha integrado en el repositorio, no se modifica, a no ser que sea necesario tal y como aclara uno de ellos:

[I2] Cuando aprendo un nuevo *idiom*, no suelo mirar donde puedo usarlo. Lo añado a mi “caja de herramientas”, y posteriormente cuando “toco” algo, lo modifico y lo dejo mejor, [...] pero no es una obsesión.

Capítulo 5

Aplicación web de análisis de proyectos Python

5.1. Introducción a la plataforma

Las herramientas actuales de análisis de código, como puede ser el caso de PEP8 o pylint, realizan un análisis de nuestro código para identificar posibles errores de sintaxis y estilo, y de esta manera poder corregirlos.

El análisis sobre el que estamos enfocando nuestro proyecto está más dirigido hacia estructuras o elementos *idiomáticos* que demuestran un mayor o menor conocimiento acerca del lenguaje. Además, estas estructuras facilitan la lectura del código (tal y como hemos analizado en las entrevistas), y permiten realizar tareas complejas en menos líneas de código.

Conocer si utilizamos los *idioms* de Python en nuestros repositorios, podría servirnos para evaluarnos en nuestra experiencia en el lenguaje, y además, obtener retroalimentación en relación a *idioms* que no estamos utilizando, y por lo tanto, aprender a utilizar los mismos.

Debido a que no existe ninguna herramienta de identificación de *idioms*, nos pareció interesante el desarrollo de una herramienta enfocada en dicho ámbito. Se basaría en una aplicación web que descargaría los repositorios de un determinado usuario, los analizaría, y finalmente ofrecería al usuario final qué *idoms* ha utilizado, y cuáles no han sido identificados.

De esta manera, cualquier usuario podría obtener una puntuación de su conocimiento en Python, y sobre todo, se podría agrupar en una misma web toda la información acerca de elementos *pythonic*.

5.2. Tecnologías utilizadas

La arquitectura de la aplicación está basada en un modelo cliente-servidor. Las tecnologías utilizadas para el desarrollo del servidor han sido *Django*, una base de datos y un módulo de Python para el análisis de repositorios. Para el cliente se han utilizado librerías de *JavaScript* como *JQuery* y para el estilo *Bootstrap*.

Django es un *Web framework* de Python utilizado para el desarrollo de servidores. En este proyecto se encarga de atender las peticiones de un determinado cliente y devolver una respuesta. Tiene una arquitectura *REST*, por lo tanto cada mensaje HTTP incluye la información necesaria para procesar la petición. En este servidor se hace un uso mínimo de *Cookies*, generadas automáticamente por *Django* para la gestión de seguridad en los formularios.

La base de datos utilizada para un despliegue en local es *SQLite*. El conjunto de datos son guardados en un único fichero, y para un conjunto de pruebas de análisis de 50 repositorios en paralelo no ha generado problemas. Para un mayor despliegue, se recomienda el uso de bases de datos con mayor seguridad y poder manejar múltiples peticiones de manera más segura. El cambio en la configuración es mínimo.

Por último, en la parte del servidor, se ha utilizado un módulo de identificación de *pythonic idioms* en repositorios. Esta aplicación fue desarrollada para el Trabajo de Fin de Grado del mismo autor de este proyecto. El módulo identificaba *estructuras* o elementos de Python considerados *pythonic*. Además se obtenía el autor del mismo a partir del correo electrónico utilizado en el *commit*. Para más información del desarrollo de esta herramienta, se puede acceder a la memoria disponible públicamente¹.

Para la implementación del lado del cliente, se han utilizado librerías *JavaScript* para ofrecer al usuario una interacción más dinámica con el servicio. Entre las librerías utilizadas destaca *JQuery*, una biblioteca que simplifica la manera de acceder a los elementos de la página, manejar los eventos, integrar animaciones y realizar llamadas *AJAX* (peticiones al servidor realizadas de manera asíncrona y en segundo plano mientras el usuario navega por la página).

Por otro lado, para el estilo de la página web, se ha utilizado *Bootstrap*. Esta biblioteca multiplataforma y de código abierto, permite realizar el diseño de una página web de un modo

¹<https://gsyc.urjc.es/~grex/pfcs/2016-jose-javier-merchante/JoseJavierMerchante-AnalisisYBusquedaDeIdiomsEnPython.pdf>

más dinámico mediante el uso de plantillas para cada uno de los elementos.

5.3. Esquema de análisis de repositorios

En la figura 5.1 se ilustra el esquema de análisis de repositorios para un usuario.

La primera etapa del análisis es la introducción de un usuario u organización de GitHub para realizar el análisis de sus repositorios. Otra opción más simplificada sería el análisis de un único repositorio, que simplificaría la explicación del proceso. En la figura 5.2 se puede ver la página de inicio.

Una vez introducido el usuario, el navegador realizará una consulta al servidor Django para obtener todo los repositorios de la cuenta del usuario.

```
GET /repos-user/<name>
```

Django realizará una consulta a la API de GitHub para recabar información. Obtendrá el listado de los repositorios del usuario, y por cada uno de ellos el lenguaje principalmente utilizado y conocer si es un *fork* (una copia del repositorio de otro usuario).

```
GET https://api.github.com/users/<name>/repos
```

Toda la información obtenido se mostrará en el navegador, para que de esta manera se pueda seleccionar los repositorios que quiere que se analicen. Por poner un ejemplo, si un usuario sabe que no hay código de Python en un determinado repositorio, no tendría sentido seleccionarlo. Además se le ofrece la opción al usuario de indicar otros repositorios a los que ha contribuido para hacer el análisis más preciso. Un ejemplo gráfico se muestra en la figura 5.3

Cuando el usuario tiene decididos los repositorios que se van a analizar, envía los datos al servidor.

```
POST /analyze-repos-user
repos: ["https://github.com/abc/cde.git", ...] username: <name>
```

Cada análisis se crea en un thread separado, y cada uno se encarga de publicar su estado en una base de datos cada cierta cantidad de datos analizados.

Los resultados de cada *idiom* se almacena en la base datos con el formato de la tabla 5.1:

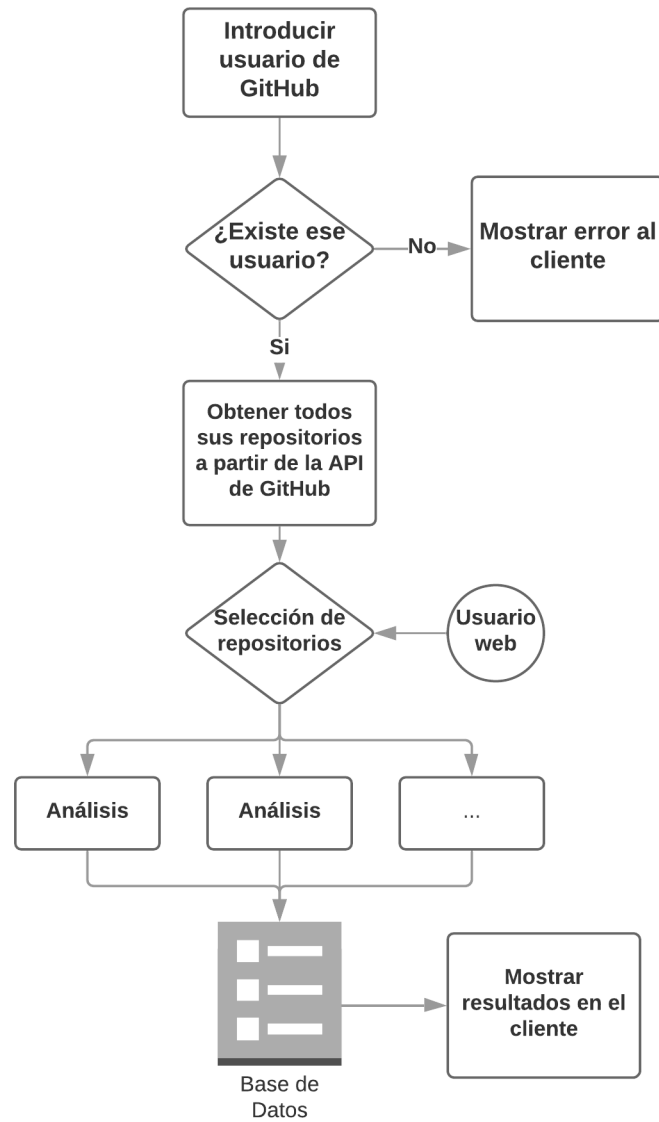


Figura 5.1: Gráfico de petición de análisis de un usuario



Figura 5.2: Página principal

Select	Name	Forked	Main language	Link
<input checked="" type="checkbox"/>	jjmerchante/2015-sat-pfinal		Python	View
<input type="checkbox"/>	jjmerchante/2017_Sattose		TeX	View
<input type="checkbox"/>	jjmerchante/C-programs		C	View
<input type="checkbox"/>	jjmerchante/Code	✘	HTML	View
<input type="checkbox"/>	jjmerchante/code_for_ITS	✘	Jupyter Notebook	View
<input checked="" type="checkbox"/>	jjmerchante/cpython	✘	Python	View
<input type="checkbox"/>	jjmerchante/CursosWeb.github.io	✘	HTML	View
<input checked="" type="checkbox"/>	jjmerchante/django-rest-framework	✘	Python	View
<input type="checkbox"/>	jjmerchante/ghtorrent.org	✘	HTML	View
<input type="checkbox"/>	jjmerchante/GORS_P4		null	View
<input checked="" type="checkbox"/>	jjmerchante/grimoirelab-perceval	✘	Python	View
<input type="checkbox"/>	jjmerchante/memoriaTFG-JoseJavierMerchantePicazo	✘	TeX	View
<input checked="" type="checkbox"/>	jjmerchante/my2048		Python	View

Figura 5.3: Selección de repositorios

Campo	Valor
repository	ForeignKey
idiomName	CharField
line	IntegerField
file	CharField
author	EmailField
method	CharField

Cuadro 5.1: Tabla de resultados en la base de datos

En la interfaz web del navegador se muestra en todo momento el progreso de cada uno de los repositorios tal y como se observa en la figura 5.4.

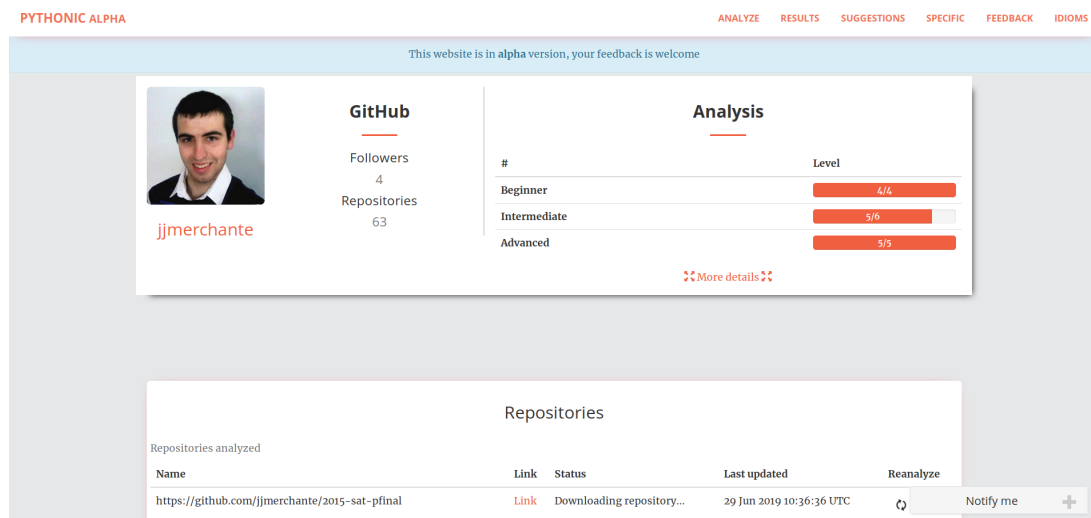


Figura 5.4: Resultados de un usuario

Cada vez se analiza un proyecto, el usuario puede acceder a la información del repositorio para ver todos los *idioms* que han sido encontrados. Una demostración visual se encuentra en la figura 5.5

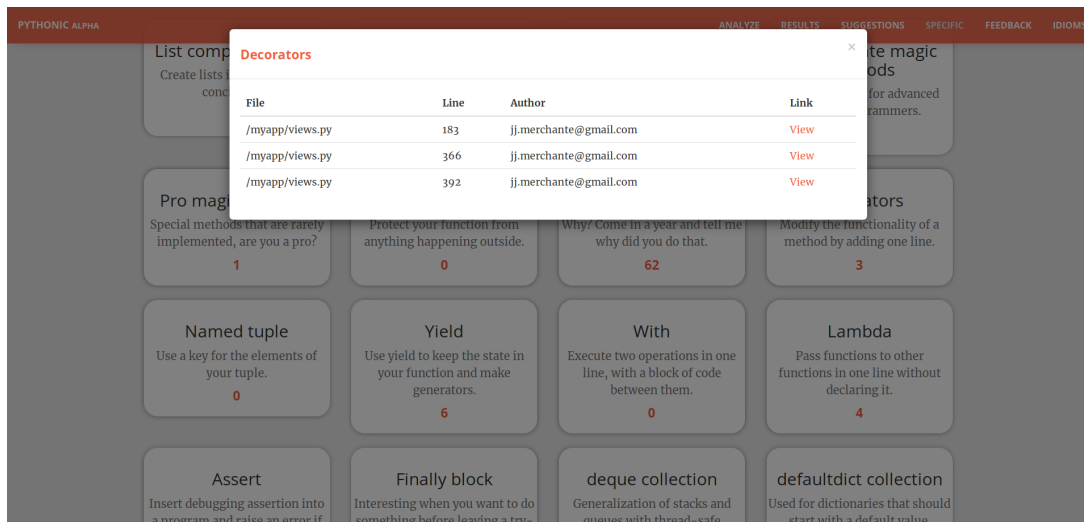


Figura 5.5: Resultados de un repositorio. Indica la localización de cada *idiom*

En ocasiones, el análisis de algunos repositorios puede ser lento debido a la cantidad de ficheros para analizar, para ese tipo de casos, en la interfaz web se ha incorporado la posibilidad de notificar al usuario mediante correo electrónico.

Por otro lado, en la página del usuario analizado, se intentan mostrar sus propios *idioms*. El manejo de identidades, en este caso, es bastante mínimo pero funcional para muchas situaciones.

Por cada uno de los *idioms* encontrados podemos conocer el autor de la línea con *git blame* y a partir de ahí obtener el correo electrónico.

El *mail* de un usuario es algo más complicado de obtener. En la mayoría de los casos no se indica su correo electrónico de manera pública, por lo tanto nos limita.

Sin embargo, podemos acceder a su historial de *commits* (tiene un alcance limitado y si no ha tenido actividad durante el último mes es probable que no se reflejen resultados) y a partir de ahí se puede obtener su correo electrónico para cruzar los datos e indicar cuáles eran sus *idioms*.

Conociendo los *idioms* que utiliza un usuario en su código, podemos saber cuáles no utiliza, de esta manera, podemos ofrecerle documentación acerca de nuevos elementos *pythonic* que podrían serle de gran utilidad para aprender.

5.4. Recopilación de *idioms* con ejemplos

Finalmente, en la misma página web se incluye un apartado con un catálogo de *pythonic idioms* que hemos considerado de mayor relevancia para mostrar a los usuarios.

Esta selección de elementos y estructuras, incluye una variedad de ejemplo para ilustrar cuándo deben usarse con referencias externas para ampliar la documentación.

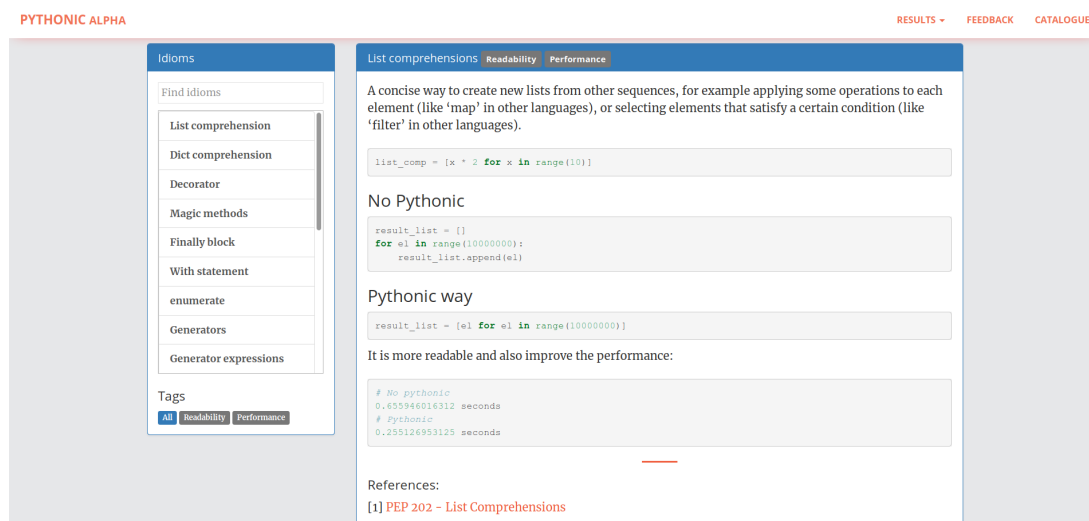
Las categorías en las que hemos dividido los *idioms* han sido por legibilidad y por rendimiento, mostrando ejemplos en ambos casos.

La lista completa de *idioms* incluidos son los siguientes:

- *List comprehensions*: manera concisa de realizar lista en una única línea a partir de otras secuencias
- *Dict comprehensions*: forma elegante de construir un diccionario de manera similar a las *list comprehensions*
- *Decorators*: permite modificar el comportamiento de una función sin alterar su código
- *Magic methods*: son métodos especiales que pueden ser utilizados al usar determinada sintaxis en Python.
- *Finally*: permite ejecutar código cuando finalice un bloque. Es común en otros lenguajes.
- *Context manager*: con *with* se pueden ejecutar un bloque de código con unas condiciones de inicio y fin. Por ejemplo útil para la apertura de ficheros.
- *enumerate*: dado un objeto iterable devuelve una tupla con el índice de cada elemento y el valor.
- *Generators*: permite la utilización de funciones como generadores de contenido sin la necesidad de mantener la información que se genera en memoria.
- *Generator expressions*: forma concisa de crear generadores similares a las *list comprehensions*.
- *defaultdict*: misma funcionalidad que la de un diccionario con la peculiaridad de que se puede definir un valor por defecto para los elementos que no están creados.
- *deque*: es una cola en ambos sentidos (cola y pila)
- *counter*: es una subclase del diccionario que permite contar objetos.

- *classmethod*: indica que el método pertenece a la clase. No necesita una instancia para ser llamado.
- *staticmethod*: no recibe la clase como parámetro al contrario que *classmethod*, “no tiene” referencias a las clase.
- *zip*: permite agrupar elementos a partir de dos secuencias diferentes.
- *itertools*: una serie de funciones que permite operar sobre iteradores.
- *total_ordering*: definiendo un método para “==” y uno diferente de comparación, permite suplir el resto para una determinada clase.

En la figura 5.6 muestra el formato con el que se representan los ejemplos.



The screenshot shows the 'Pythonic Alpha' website interface. On the left, there is a sidebar with a search bar and a list of idioms: List comprehension, Dict comprehension, Decorator, Magic methods, Finally block, With statement, enumerate, Generators, and Generator expressions. The main content area is titled 'List comprehensions' and includes a description: 'A concise way to create new lists from other sequences, for example applying some operations to each element (like 'map' in other languages), or selecting elements that satisfy a certain condition (like 'filter' in other languages)'. Below this, there are three sections: 1. A code example: `list_comp = [x * 2 for x in range(10)]`. 2. 'No Pythonic': A code example showing a for loop and append: `result_list = []
for el in range(10000000):
 result_list.append(el)`. 3. 'Pythonic way': A code example showing a list comprehension: `result_list = [el for el in range(10000000)]`. Below the code examples, there is a performance comparison table:

Method	Time
No pythonic	0.655946016312 seconds
Pythonic	0.255126953125 seconds

. At the bottom, there are 'References' including a link to 'PEP 202 - List Comprehensions'.

Figura 5.6: Explicación de *list comprehension* en la web

Capítulo 6

Conclusiones

En este capítulo se indican las conclusiones y resultados finales obtenidos tras la realización de este proyecto. La consecución de los objetivos ha sido prácticamente total. No obstante, siempre existe un determinado margen de mejora sobre el que se puede hacer un estudio intenso. En este apartado además se mencionan lecciones aprendidas a lo largo del proyecto y se concluye con una valoración personal acerca del desarrollo del mismo.

6.1. Consecución de objetivos

Como en todo proyecto es siempre importante analizar con detalle qué logros han sido conseguidos y analizar el desarrollo del trabajo realizado. Por todo ello, es importante debatir los objetivos logrados del Capítulo 2 de manera objetiva.

Este proyecto se divide en dos partes principalmente diferenciadas. Por un lado se encuentra la realización de entrevistas a programadores de Python con el objetivo principal de recabar información acerca del lenguaje.

Las entrevistas fueron realizadas a trece candidatos, en principio el número puede parecer escaso, sin embargo, atendiendo a la variedad de perfiles, y a la correlación entre ellos, parece también justo afirmar que las conclusiones obtenidas podrían haber sido bastante similares de tener un mayor número de candidatos.

Las respuestas obtenidas, atendiendo a los objetivos, nos dieron una idea bastante precisa acerca de qué opina la comunidad que programa con Python acerca de los elementos *pythonic*. Suelen considerarlos estructuras que demuestran la madurez de un programador en el lenguaje,

suelen facilitar la lectura del código y en algunas ocasiones mejoran el rendimiento del mismo.

Por otro lado, hemos notado cómo la importancia que dan para las empresas que publican su código en abierto al código *pythonic* es superior a las empresas en código cerrado, no obstante, quizá es una opinión sesgada, debido a que la mayor parte de candidatos publicaban su código.

La manera en que se transmiten los *idioms* por la comunidad tiende a ser variada, principalmente es leyendo código de otros programadores, sin embargo, en muchas ocasiones es el interés que despierta el lenguaje. En Python, siempre existe una manera buena de hacer las cosas [4], por lo tanto, aprender esa metodología siempre te va a hacer mejor programador.

En la parte del servicio web, siento haber conseguido integrar el software de mi TFG para que en este caso fuera más accesible para cualquier usuario. Ofrecer datos más enriquecidos siempre motiva más al usuario final.

Por otro lado incluir un catálogo de *idioms* a la web me pareció un acierto debido a que integra el aprendizaje en un único lugar.

No obstante, hubo una parte que no logré integrar a la perfección. El análisis de *anti-idioms* y la detección de los mismos. Podría considerarse como una mejora del mismo para ofrecer un mayor nivel de análisis de repositorios. Por el momento la web es un método de aprendizaje, quizá la detección de *anti-patrones* pueda incorporarse en mejor medida en un IDE donde el usuario puede identificar la mejora en tiempo real, y no tras subir su código a un repositorio.

En definitiva, prácticamente todos los objetivos propuestos al inicio de este proyecto han sido abordados y superados con éxito.

6.2. Lecciones aprendidas

A lo largo del proyecto, he podido adquirir nuevos conocimientos y experiencias que no se imparten directamente en el máster. La realización de entrevistas puede parecer algo relacionado más directamente con carreras de ciencias sociales, en cambio, leyendo documentación y artículos, he podido comprobar que existe una gran cantidad de proyectos, sobre todo de investigación, que precisan de una parte más social.

Preguntar a un determinado colectivo te aporta una gran información, que quizá buscando en la web puede parecer sesgada por tus búsquedas anteriores. Así mismo la realización de entrevistas me ha enseñado que cada entrevista es distinta a la anterior, cada persona puede

orientarla hacia su punto de vista a pesar de que las preguntas vayan en un guión.

Por otro lado, en relación a realizar el servicio web, la complicación no fue demasiado elevada, pero siempre te encuentras con problemas de los que acabas aprendiendo un poco más.

Considero importante este trabajo de fin de máster con el que he podido obtener nuevo conocimiento, quizá más enfocado a un ámbito más de investigación, pero que genera una nueva visión en esta nueva etapa

6.3. Trabajos futuros

Es importante señalar que todo proyecto puede mejorarse, se pueden integrar nuevas funcionalidades o mejorar de cara a un usuario. El estudio de análisis de *idioms* es un campo que tiene mucho recorrido, es por ello que en este epígrafe se señalan algunos puntos para futuros trabajos:

1. Podría ser interesante crear un *plug-in* para un IDE que permita identificar *anti-idioms* y que el usuario pueda detectarlo al momento. De este modo se podría mejorar la legibilidad del código desde el primer momento.
2. Para la página web podría haber mejoras en relación al manejo de identidades para identificar el autor de un *idiom*. Podrían mostrarse al usuario agrupaciones de correos o nombres para que las pueda relacionar.
3. Para realizar un mejor estudio de Python se podrían realizar encuestas simples con fragmentos de código para saber qué es lo que consideran más o menos *pythonic*.
4. Otro posible análisis sería identificar la evolución de *idioms* a lo largo del tiempo sobre una muestra considerable de repositorios con alta actividad durante un largo periodo de tiempo.

Apéndice A

Preguntas realizadas en las entrevistas

Las principales preguntas realizadas a los entrevistados son las que aparecen en la siguiente lista. El orden pudo haber cambiado en dependencia a las respuestas dadas, algunas pudieron ser respondidas previamente y otras fueron espontáneas para conocer algún detalle más en concreto.

- ¿Eres familiar con el término *pythonic*? ¿Qué es para ti?
- ¿Podrías darnos algún ejemplo?
- En tu opinión, ¿el código *pythonic* es importante? ¿Es el concepto de código *pythonic* diferente de la necesidad de un estilo de código en otros lenguajes? ¿Cuáles son las principales diferencias?
- ¿Crees que es deseable? ¿Qué características crees que son más importantes? ¿Es un programador más avanzado por el uso de *idioms* de Python?
- ¿Crees que escribir código *pythonic* es más un estilo o un conjunto de recetas? ¿Podrían ser catalogados?
- ¿Cuando programas piensas en usarlos [*pythonic idioms*] o no piensas acerca de ellos?
- ¿Crees que tu código es más *pythonic* con los años? ¿Por qué?
- ¿Entendías bien los *idioms* cuando empezaste a aprender? ¿Cuáles te parecieron más complicados? (En algunos casos se mencionaron algunos si no se acordaba de ninguno)

- ¿Cómo aprendiste a programar más *pythonic*? (En algunos casos se preguntó detalles acerca de conferencias, StackOverflow, compañeros o libros)
- ¿Qué IDE usas habitualmente? ¿Utilizas alguna extensión o librería para Python?
- En el trabajo, ¿cómo de recomendable es el código *pythonic*? ¿Es obligatorio o es una elección personal?
- ¿Comparas el código con tus compañeros? ¿Os revisáis el código en términos de estilo?
- ¿Subes el código a una plataforma de software abierto?
- ¿Crees que programar *pythonic* en una entrevista puede resultar beneficioso?
- ¿Podrías ser capaz de diferenciar un programador experto de un principiante por su código? ¿Cómo?
- Cuando descubres un nuevo *idiom*, ¿adaptas tu antiguo código?

También se mostraron a algunos entrevistados (los que fueron en persona) una lista con una serie de *idioms* para identificar diferentes aspectos para la mejora de la herramienta de detección del nivel de un programador Python.

Bibliografía

- [1] C. V. Alexandru, J. J. Merchante, S. Panichella, S. Proksch, H. C. Gall, and G. Robles. On the usage of pythonic idioms. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2018*, pages 1–11, New York, NY, USA, 2018. ACM.
- [2] S. Kvale. *Interviews: an introduction to qualitative research interviewing*. Sage Publications, 1996.
- [3] S. Merriam. *Qualitative Research: A Guide to Design and Implementation*. Higher and adult education series. John Wiley & Sons, 2009.
- [4] T. Peters. The zen of python. In *Pro Python*, pages 301–302. Springer, 2010.
- [5] G. van Rossum, B. Warsaw, and N. Coghlan. *PEP 8: style guide for Python code*. Python.org, 2001. <https://goo.gl/crVen9>.