Aprende programación con

Python

Aprende programación con

Python

Desarrolla tus propias aplicaciones

Jose Vicente Carratalá Sanchis

Dedicado a todas aquellas personas que conozco que destinan sus esfuerzos a crear un mundo mejor, de forma consciente o sin pretenderlo, con grandes o pequeñas acciones.

Tabla de contenidos

1. Prólogo

Las máquinas no son importantes: Las personas sí que lo son. Cuando aprendemos a crear aplicaciones informáticas, debemos pensar en cuál es nuestro objetivo al crear dichas aplicaciones - y si no tenemos un objetivo definido, quizás ahora es un buen momento para tenerlo.

En muchas ocasiones el lector tendrá un interés mayormente concreto con respecto al aprendizaje de creación de aplicaciones informáticas. Probablemente dicho interés tiene que ver con objetivos de progreso profesional. Y efectivamente este libro está escrito con esos objetivos en mente: habilitar al lector para que sea capaz de desarrollar aplicaciones informáticas de todo tipo y de forma autónoma.

Sin embargo, al comienzo de esta obra, me gustaría compartir contigo mi visión acerca del desarrollo de aplicaciones informáticas, por si tal vez mi punto de vista pueda enriquecer al tuyo, y te pueda dar una perspectiva más amplia de las habilidades que, a continuación, vas a aprender.

Vivimos en un mundo en el que los sistemas informáticos son omnipresentes, y las personas dependen cada vez más de estos sistemas, para prácticamente todas las facetas de su vida. El objetivo de este prólogo no es opinar acerca de si esto es bueno o no lo es, sino meramente constatar que es así, y que, al parecer, no solo seguirá siendo así, sino que irá a más.

En este momento en el que los programas informáticos influyen en muchos aspectos de nuestra vida, los desarrolladores de aplicaciones tenemos no solo el poder de influir en las vidas de las personas, sino también el deber de usar esa responsabilidad de forma correcta.

En el presente, nuestras aplicaciones pueden cambiar la vida de las personas que las utilizan. Si las diseñamos bien, permitirán a sus usuarios realizar más trabajo en menos tiempo, y por lo tanto, tendrán más tiempo para sí mismos, o para su ocio, o para sus familias.

En el presente, nuestras aplicaciones pueden cambiar el mundo en el que vivimos, y en el que viviremos en el futuro. Aprendiendo a programar, aprendiendo a crear programas informáticos, se pone en tus manos más poder del que puedas pensar en un primer momento.

Lo que tienes que preguntarte es: ¿qué vas a hacer con ese poder?

2. Introducción

2.1. El objetivo de este libro

Este libro no pretende ser una guía exhaustiva sobre el lenguaje de programación Python. Es, sin embargo, una forma de guiarte paso a paso en el proceso de aprendizaje de programación en este lenguaje, con el objetivo de descubrirte una a una las piezas fundamentales que te van a permitir crear tus propios programas.

No encontrarás, en este libro, un listado de comandos y opciones más exhaustivo que el que vas a encontrar en la propia documentación oficial del lenguaje, que puedes encontrar aquí:

https://docs.python.org/3/

Sin embargo, lo que sí que vas a encontrar en este libro es justo lo que no te proporciona la documentación: una explicación pausada, usando terminología clara para las personas que no son programadoras, una guía y un recorrido por las funciones clave, y una serie de ejercicios comentados que te permitirán entender los fundamentos del lenguaje y las herramientas de las que se compone, y entonces, si quieres, es cuando podrás acudir a la documentación oficial para profundizar en el conocimiento de cada una de las instrucciones del lenguaje.

2.2. Sobre el autor

Jose Vicente Carratalá es un profesor, desarrollador y diseñador, originario de Valencia, España.

Su labor profesional se inicia en el año 2000, al finalizar su formación universitaria, cuando comienza su carrera como formador, impartiendo cursos de formación de postgrado en la Universidad Politécnica de Valencia.

A lo largo de los años ha sido formador en multitud de modalidades y niveles, impartiendo formación universitaria, ciclos formativos, y cursos especializados para empresas, llegando a colaborar en destacados centros

de formación online como video2brain y Linkedin Learning, entre otros muchos.

Desde 2006 compagina la docencia con los proyectos que desarrolla en su propio estudio: JOCARSA, cuyo objetivo es crear desarrollos personalizados para clientes combinando las tecnologías más punteras en cuanto a desarrollo de aplicaciones informáticas y técnicas de visualización 2D y 3D.

Puedes encontrar más información sobre el autor en:

https://jocarsa.com

2.3. Breve historia de los lenguajes de programación

Los lenguajes de programación surgen con la aparición de las máquinas de computación programables: máquinas que no desarrollaban una única función, sino que podían ser reprogramadas mediante datos introducidos por los usuarios.

En base al sistema binario con el que funciona la práctica totalidad del hardware informático moderno, el lenguaje binario es la forma última de proporcionar instrucciones a la máquina - pero, para el ser humano, resulta extremadamente arduo expresar su voluntad en este lenguaje.

Por esto, en la segunda mitad del siglo XX, se desarrollaron una serie de lenguajes a diferentes niveles de abstracción, para permitir al ser humano programar en un lenguaje más similar al lenguaje humano - donde, más adelante, otro programa llamado intérprete o compilador se encarga de traducir las instrucciones dadas por el ser humano en el lenguaje binario con el que trabaja la máquina.

Muchos de estos lenguajes, sin embargo, todavía no presentaban un nivel de abstracción de la complejidad suficientemente grande como para poder atraer, al mundo de la programación, a profesionales de cualquier sector.

2.4. Breve historia de Python

Python es un lenguaje de programación de alto nivel, esto quiere decir que mucho de su funcionamiento está abstraído para resultar más fácil de usar por parte del usuario final.

La primera versión oficial se lanzó en 1991. Python 2 se lanzó en el año 2000, y Python 3 en el año 2008. Teniendo en cuenta los problemas de retrocompatibilidad que ha habido entre Python 2 y Python 3, que han causado que las dos versiones convivan paralelamente durante casi una década (algo muy poco frecuente en el terreno de los lenguajes de programación), en el momento de escribir este libro no parece haber planes activos de desarrollo para una versión 4 del lenguaje.

El objetivo original de Python consiste en habilitar al usuario a realizar las mismas tareas que pueden realizarse en otros lenguajes de programación, pero simplificando las instrucciones y utilizando mucho menos código.

Por otra parte, Python es un lenguaje que incorpora una gran cantidad de funcionalidades en su instalación de base, y en el que es muy sencillo instalar nuevas librerías a demanda.

Por este motivo, Python se ha convertido en el lenguaje más solicitado actualmente en el sector informático, incluso por delante de otros lenguajes como C, C++ o Java.

Python es un lenguaje muy atractivo para muchos tipos de profesionales, no necesariamente familiarizados previamente con el terreno de la programación de aplicaciones, ya que se basa en una serie de reglas muy sencillas para escribir código.

2.5. Características de Python

2.5.1. Lenguaje interpretado

A diferencia de otros lenguajes, como el lenguaje C, Python no genera archivos compilados en lenguaje binario dependiente de la plataforma y de

la arquitectura, sino que los archivos de código de Python son, esencialmente, archivos de texto, que se ejecutan contra el intérprete de Python, que adapta el código, durante su ejecución, a las características de la máquina en la que se está ejecutando.

2.5.2. Multiplataforma

Una de las características principales de Python es que es multiplataforma. Sus programas se pueden ejecutar en cualquier sistema operativo, en cualquier plataforma, y en cualquier arquitectura, que cuente con un intérprete de Python - y son muchas.

Refiriéndonos a las tres plataformas PC principales (Windows, macOS y Linux), esta característica de Python nos asegura poder desarrollar una aplicación informática, y poder ejecutarla en los tres sistemas de forma simultánea sin tener que hacer desarrollos independientes para cada plataforma - lo que simplifica enormemente el proceso de desarrollo.

2.5.3. Multiparadigma

Python implementa diferentes paradigmas de desarrollo de software, para que cada programador utilice aquel con el que se sienta más cómodo, y/o el que se adapte mejor a cada situación. Los lenguajes multiparadigma, de hecho, actualmente dominan los listados de lenguajes de programación más utilizados.

2.5.4. Lenguaje de alto nivel

Los lenguajes de programación de bajo nivel son aquellos que están más cerca del “lenguaje máquina”, y los lenguajes de programación de alto nivel son aquellos que están más cerca del “lenguaje humano”. Dentro de esta clasificación, Python se sitúa claramente (al menos comparativamente con respecto a la globalidad de los lenguajes de programación) en el lado de aquellos lenguajes que realizar abstracciones para que la sintaxis resulte lo más intuitiva posible para el usuario promedio

2.5.5. Indentación funcional

La indentación, o sangría, es el número de espacios que contiene cada una de las líneas de nuestro programa. Python es uno de los pocos lenguajes en los que la sangría no cumple una finalidad meramente estética, sino que define el nivel de jerarquía o pertenencia a un bloque de cada una de las líneas. Debemos tener cuidado con el uso de la indentación en nuestros programas, puesto que su mal uso puede afectar al funcionamiento de nuestro programa.

2.6. El Zen de Python

En Zen de Python consiste en una serie de guías filosóficas que tienen como objetivo orientarnos a la hora de crear programas con este lenguaje de programación - a decir verdad, son extrapolables a otros muchos lenguajes.

Bello es mejor que feo.

El concepto de “bello” es muy específico para lo que es un lenguaje de programación, con respecto a lo que es la belleza en general. En programación se entiende que algo es bello cuando está estructurado, proporcionado, espaciado, comentado, y en definitiva cuando la presentación del código invita a su lectura y facilita su comprensión.

Explícito es mejor que implícito.

Python es un lenguaje pensado para ser entendible incluso por personas que no son programadoras. Ante la elección, el programador debe elegir soluciones que sean explícitas, aunque usen más líneas de código, o usen recursos innecesarios, desde el punto de vista del ahorro de recursos, tales como “variables intermediarias”.

Simple es mejor que complejo.

Si para implementar una funcionalidad, el código resultante es complejo, deberíamos preguntarnos si es el mejor código que podríamos escribir. El código en Python, por la propia naturaleza de este lenguaje de programación, debería ser lo más sencillo posible.

Complejo es mejor que complicado.

Existen ocasiones en las cuales no es posible simplificar el código, y la propia naturaleza de la solución informática o el algoritmo que aplicamos requiere escribir código complejo. Cuando esto ocurre, siempre es preferible que la solución sea compleja, pero entendible, a complicar artificialmente la solución.

Plano es mejor que anidado.

Las estructuras de control nos ofrecen la posibilidad conveniente de anidar unas estructuras dentro de otras. Sin embargo, anidar estructuras, especialmente cuando se hace de una forma abusiva, va en contra de la legibilidad del código, y en ocasiones, también va en contra de la eficiencia en la ejecución.

Es por esto que, siempre que se pueda evitar, debe intentarse que la ejecución del código no presente anidaciones abusivas ni injustificadas.

Espaciado es mejor que denso.

Cuando empezamos a escribir en un lenguaje de programación concreto, generalmente utilizamos un estilo de escritura espaciado, en el cual dividimos todo aquello que hacemos en diferentes líneas de código.

A medida que vamos profundizando en nuestro conocimiento acerca de ese lenguaje de programación, es frecuente escribir un código más denso, en un intento de ahorrar algo de esfuerzo en la escritura del código. El problema es que el código denso no es tan legible como el código espaciado, y la legibilidad del código es uno de los objetivos principales dentro de Python. Por lo tanto, aunque nuestro conocimiento del lenguaje sea extenso, deberemos intentar escribir siempre el código espaciado y evitar escribir código denso.

La legibilidad es importante.

Este es uno de los principios más sencillos de seguir, el código que escribamos debe ser lo más claro y lo más diáfano que sea posible. Debemos cuidar los espacios, la nomenclatura, y en definitiva toda la escritura del código en general para que nuestros programas resulten legibles.

Los casos especiales no son lo suficientemente especiales como para romper las reglas.

Existen una serie de reglas, como este mismo zen, o como conjuntos de reglas de buenas prácticas. La idea es intentar ceñirnos todo lo posible a esos principios básicos de funcionamiento, y no salirnos de esos principios a la primera oportunidad o dificultad que se nos presente.

Sin embargo la practicidad le gana a la pureza.

Sin embargo, los principios teóricos de buenas prácticas son meras intenciones y no leyes, y la realidad del desarrollo de las aplicaciones informáticas consiste en que pueden existir situaciones que requieran romper las reglas. Si la situación lo justifica y el software lo requiere, se pueden romper las reglas si ello conlleva claros beneficios en el desarrollo de una solución de software concreta.

Los errores nunca deberían pasar silenciosamente.

Como desarrolladores tenemos la responsabilidad de que nuestro código funcione correctamente. Para ello, muchas veces, en porciones de nuestro código, tenemos bloques que, en determinadas circunstancias, y en determinados escenarios, pueden generar errores. Llegados a este caso, podemos actuar de muchas formas diferentes.

En todo caso nunca debemos esconder los posibles errores que dé nuestro programa, porque esta práctica se nos puede volver en nuestra contra cuando el proyecto crece. Debemos intentar prever y atrapar todos los errores, mediante las estructuras de control que se van a presentar para ese fin dentro de este libro.

A menos que se silencien explícitamente.

Sin embargo, la realidad del desarrollo de un programa muchas veces requiere tratar de forma silenciosa posibles errores que se puedan llegar a dar.

Frente a la ambigüedad, evitar la tentación de adivinar.

En muchas ocasiones, cuando un código falla, muchos programadores intentan adivinar dónde está el fallo desactivando y activando partes del programa, hasta que todo parece funcionar de nuevo. Debemos evitar esta metodología, analizar dónde está realmente el problema, y proporcionar una solución efectiva.

Debería haber una, y preferiblemente sólo una, manera obvia de hacerlo.

Frente a otros lenguajes de programación que animan a considerar la existencia de varios caminos para resolver un problema, el creador de Python anima, mediante esta parte del Zen de Python, a encontrar una forma óptima de resolver cada problema o cada situación que se presente.

A pesar de que eso no sea obvio al principio a menos que seas Holandés.

Desde hace mucho ha existido una corriente de discusión en internet acerca del significado de este principio filosófico. Posiblemente el "holandés" hace referencia al científico de datos Edsger W. Dijkstra, defensor de un principio de desarrollo contrario al anterior, en el cual se indica que probablemente hay varias formas de hacer una misma tarea - otras fuentes señalan que, dado que Guido Van Rossum, el creador de Python, es también holandés, este principio filosófico es una referencia y un guiño cultural a todos los holandeses implicados en el desarrollo de Python.

Ahora es mejor que nunca.

Creo que este es mi principio zen favorito. Vendría a ser similar que ese otro principio que dice que "hecho es mejor que perfecto", o que "la optimización prematura es la raíz de todo mal". La idea es que hay veces en las que planificar demasiado hace que las cosas nunca se hagan, y por lo tanto, es mejor hacer cosas que sean imperfectas, que nunca hacer algo supuestamente perfecto.

A pesar de que nunca es muchas veces mejor que *ahora* mismo.

Sin embargo, a veces antes de empezar a programar una solución informática es bueno tomarse un poco de tiempo para pensar si el camino que se va a tomar es el correcto, o quizás nos llevará a un callejón sin salida.

Si la implementación es difícil de explicar, es una mala idea.

Este es un principio que se aplica fácilmente no solo a Python, sino a prácticamente cualquier otro lenguaje de programación: si algo es difícil de explicar, con toda probabilidad, o es una mala idea, o es simplificable

Si la implementación es fácil de explicar, puede que sea una buena idea.

Sin embargo, por otra parte, que una implementación sea fácil de explicar, no quiere decir que automáticamente sea buena idea - aunque probablemente lo sea.

Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Es importante definir correctamente el ámbito de los recursos, para poder controlar la visibilidad de las mismas, con el objetivo de que unos recursos no causen colisiones con otros.

2.7. Instalación de un entorno de desarrollo

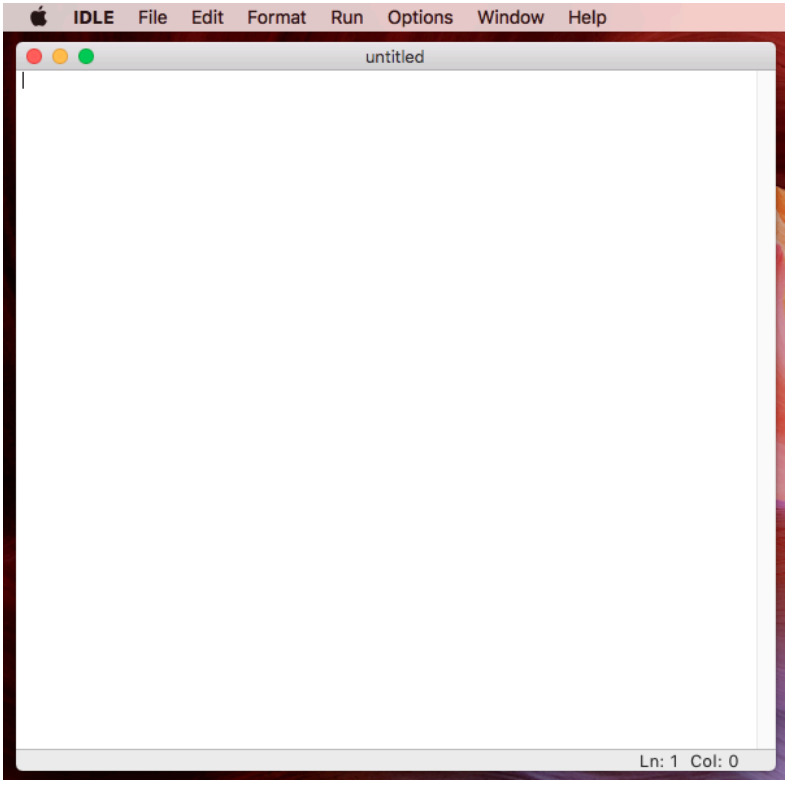
2.7.1. Edición de archivos en archivos de texto plano

Un script de Python reside en un archivo de texto plano que tiene la extensión .py. En base a esta premisa, podemos escribir un archivo en Python con un simple editor de texto, literalmente.

2.7.2. El editor IDLE

Habiendo dejado claro esto, existen Entornos integrados de desarrollo, o Integrated Development Environments, que son programas cuya misión es integrar todas las herramientas necesarias para desarrollar programas de una forma sencilla.

Con la propia instalación de Python se adjunta un editor de código llamado IDLE (en honor a Eric Idle, de Monty Python), que es el editor que recomiendo para realizar los ejercicios de este libro y para trabajar con normalidad. El editor IDLE dispone de un conjunto de herramientas mínimas (como la propia esencia de Python), y la capacidad de ejecutar automáticamente scripts de Python pulsando la tecla F5.



2.7.3. Otros IDE's

Además de IDLE, existen otros Entornos integrados de desarrollo de terceros, tales como Visual Studio de Microsoft, que proporcionan soporte para trabajar con Python.

2.8. Acerca de las capturas en este libro

2.8.1. Python como lenguaje multiplataforma

Una de las cosas que, sin duda, más me gustan de Python es su naturaleza multiplataforma, que posibilita que un mismo escrito sea ejecutado de una

forma transparente en el intérprete de Python de diferentes sistemas operativos de escritorio, Tales como por ejemplo Windows, Linux, o macOS.

En este libro se presentan, cuándo es necesario, una serie de Capturas realizadas, en mi caso, en un ordenador que tiene el sistema operativo macOS.

De esta forma, podremos reconocer en dichas capturas que la decoración de las ventanas corresponde al sistema operativo de Apple.

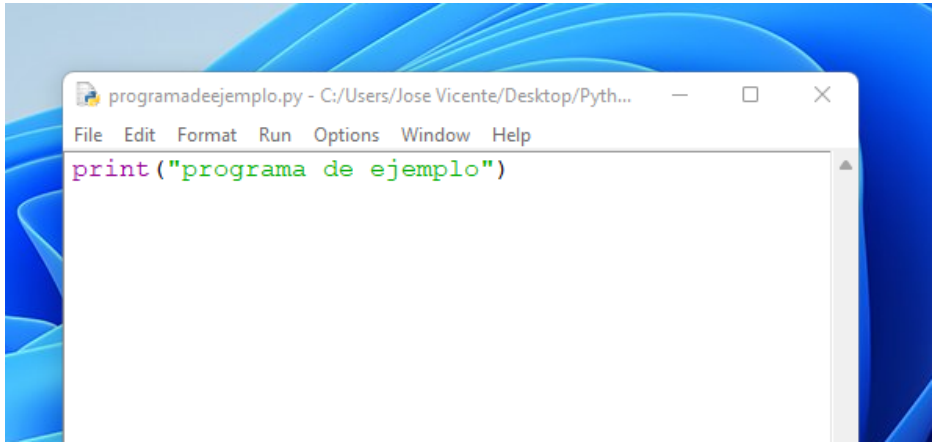
La decoración de esas ventanas, dentro de las capturas de este libro es meramente ilustrativa, y en ningún caso, quiere decir que ninguno de los ejercicios que se presentan en esta publicación, están orientados hacia el terreno Apple, o se requiera del mismo sistema operativo que tengo yo para que los ejercicios funcionen.

Más bien al contrario, todos los ejercicios que presento son totalmente compatibles con cualquier sistema operativo mayoritario de escritorio.

2.8.2. Decoraciones de ventana

2.8.2.1. Decoraciones en Windows

En el caso de que ejecutemos nuestros programas en el sistema operativo, Microsoft Windows, la decoración de las ventanas que podremos encontrar, será similar a la siguiente captura.



En el ejemplo anterior se muestra una captura correspondiente a la decoración de ventanas, por defecto en Windows 11.

2.8.2.2. Decoraciones en macOS

En el caso de que estés utilizando uno de los sistemas operativos de escritorio de Apple, dentro de la familia de sistemas operativos macOS, la decoración de ventanas que tendrás será parecida a aquella que puedes ver en las capturas de este libro.

2.8.2.3. Decoraciones en Linux

Existen una gran cantidad de distribuciones del sistema operativo Linux, y existen también muchos tipos diferentes de decoraciones de ventanas, con lo cual es difícil prever, en el caso de que uses este sistema operativo, la decoración de ventanas que tendrá tu instalación concreta.

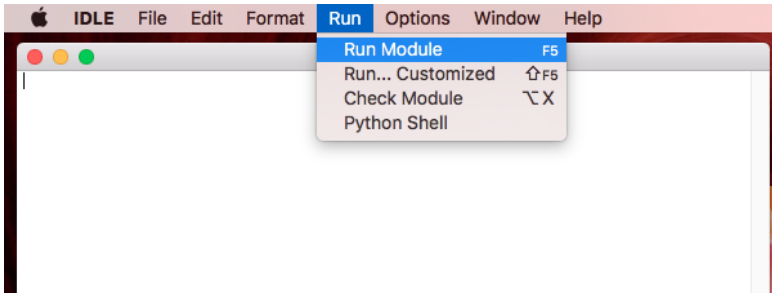
2.9. Ejecución de los scripts de Python

2.9.1. Ejecución dentro del entorno de desarrollo

La forma más cómoda de utilizar Python y, desde luego, de ejecutar sus scripts, consiste en utilizar un entorno de desarrollo integrado, o IDE, que no es más que un software que contiene las herramientas para escribir el código necesario, y la automatización para poder ejecutarlo de forma cómoda fluida.

En todo momento voy a recomendar utilizar el IDE que viene incorporado un estándar por defecto de Python, que es un programa llamado Idle.

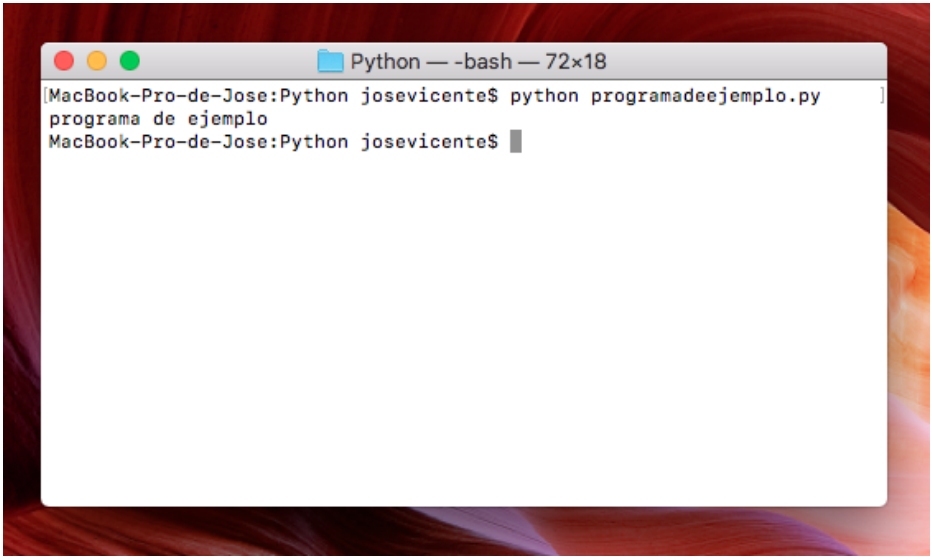
Es un programa tremendamente sencillo, que nos permite escribir código, y a continuación ejecutarlo pulsando la tecla F5, o haciendo clic en el siguiente menú.



2.9.2. Ejecución en la consola/terminal

Otra forma de ejecutar nuestros scripts consiste en llamarlos directamente desde la consola.

En el caso de que hayamos utilizado otro programa, como un editor sencillo de texto, para trabajar nuestros scripts, debemos saber que podemos ejecutarlos, simplemente llamándolos desde la consola o terminal, para lo cual, en primer lugar, deberemos navegar hasta la carpeta, donde esté alojado el script, y a continuación introduciremos el siguiente comando en la consola.



2.9.3. Plataformas soportadas

2.9.3.1. Windows

Windows es, actualmente, la plataforma informática y el sistema operativo más utilizados, y aunque en los últimos lustros está poco a poco en declive con respecto a otros sistemas operativos, tanto de escritorio como móviles, todavía sigue siendo una plataforma primaria en cuanto a equipos de escritorio.

Por lo tanto, lo más probable es que acabemos desarrollando programas informáticos encima de la plataforma *Windows*, y los usuarios de nuestras aplicaciones tengan instalado *Windows*. Por lo tanto, deberemos tener en cuenta esta circunstancia cuando creamos nuestras aplicaciones, para asegurar que sean compatibles con el sistema operativo de *Microsoft*.

2.9.3.2. Mac

El segundo sistema operativo más común en cuanto a ordenadores de escritorio y portátiles hoy en día es *macOS* de *Apple*. En las versiones más

antiguas debemos tener en cuenta que la versión que viene preinstalada es la 2.7, lo cual puede causar problemas, incompatibilidades y colisiones con la versión 3.11

2.9.3.3. Linux

Prácticamente cualquier plataforma *Linux* actual viene con Python preinstalado puesto que se requiere para la operativa del propio sistema operativo - y si no viene preinstalado, puede instalarse fácilmente descargándolo e instalándolo desde los repositorios oficiales del sistema operativo.

Para comprobar si el sistema operativo *Linux* en el que nos encontremos tiene soporte para Python (lo tiene instalado), simplemente tenemos que abrir un terminal y escribir "python -v", tras lo cual el sistema nos indicará si está instalado, y si está instalado, nos dirá la versión.

2.9.3.4. Otras plataformas

Python dispone de soporte, de forma más o menos oficial, para otras plataformas, además de las que anteriormente he mencionado.

De esta forma, nos proporciona soporte tanto para diferentes distribuciones del sistema operativo Unix, como, de forma no tan oficial, y a través de aplicaciones y complementos, para desarrollar en Python, utilizando los sistemas operativos móviles más comunes hoy en día, que son *Android* e *iOS*

2.10. Coloreado del código

En esta publicación se usan una serie de estilos al presentar el código, que incluyen el uso de coloreado, negrita y cursiva. Por ejemplo, los comentarios aparecen marcados en gris y en cursiva.

El estilo de esta publicación no tiene por qué coincidir con los estilos visuales del IDE que estés utilizando. Por lo tanto, no te preocupes de si tus estilos no coinciden con los de esta publicación.

2.11. Nomenclatura en los identificadores

A lo largo de esta publicación voy a usar convenciones de nomenclatura que son ampliamente aceptadas a través de diferentes lenguajes de programación. Su uso se recomienda, pero no es obligatorio, y desde luego no afecta al funcionamiento de los propios programas, sino a la claridad del código.

2.11.1. Variables

Las variables y estructuras de control, y las propiedades de las clases deberían empezar con minúsculas, y estar representadas por sustantivos sobre todo, y por adjetivos.

```
edad = 44
```

2.11.2. Funciones y métodos

Las funciones y los métodos de las clases deberían estar representadas por un verbo, y opcionalmente un complemento. En el caso de estar formada por dos o más palabras, se recomienda usar la nomenclatura camelCase, aunque el uso de guiones bajos también es aceptable. Deberían empezar siempre con minúsculas.

```
ponNombre()
Persona.saluda()
```

2.11.3. Clases y objetos

Las clases y los objetos derivados de esas clases deberían empezar siempre con mayúscula, y suelen estar nombradas en base a sustantivos

```
Persona = new Persona()
class Persona:
```

2.12. Mayúsculas y minúsculas

Además de lo que he comentado anteriormente, en cuanto a las buenas prácticas, con respecto a la nomenclatura de estructuras de datos, clases, y otros elementos, debemos también tener en cuenta que este lenguaje de programación, al igual que ocurre con otros muchos lenguajes de programación, es sensible a las mayúsculas y las minúsculas, entendiendo, por lo tanto, que esta variable

```
miedad = 44
```

Es, para el lenguaje de programación, internamente, una variable completamente diferente a esta otra variable.

```
miEdad = 44
```

Por lo tanto, además de respetar las buenas prácticas, en cuanto a las ocasiones en las que debemos comenzar nuestras nomenclaturas con mayúsculas y minúsculas, también deberemos conocer esta circunstancia que presento en este capítulo, para usarla, como sea debido en cada momento.

2.13. Nombres en español

Python, como la inmensa mayoría de los lenguajes de programación, tiene sus instrucciones escritas en inglés. Aunque esto parezca una desventaja para los hispanoparlantes, yo siempre lo he visto como una oportunidad, al menos desde el punto de vista didáctico: Cuando yo escriba código, verás que pongo algunas palabras en español y otras palabras en inglés. Cuando eso ocurra, la regla que podrás seguir será muy sencilla: Todo aquello que esté en español, son palabras que puedes cambiar. Y todo aquello que esté en inglés, son palabras reservadas del propio lenguaje de programación, y no las puedes cambiar.

Así pues, si en el código aparece escrito:

```
class Persona:
```

sabrás que la palabra class es una palabra reservada del lenguaje de programación, y por tanto es obligatorio que la pongas exactamente igual que yo la he escrito, pero "Persona" es un nombre que yo le he dado a esa clase (en el ejemplo que acabo de poner), y por tanto tú puedes poner cualquier otro nombre.

2.14. Uso de acentos y caracteres especiales

Ya hace años que Python, como la gran mayoría de lenguajes de programación, tiene soporte para Unicode. De esta forma, en teoría, en Python podemos usar eñes, acentos, y otros caracteres que pueden usarse dentro del idioma español, para escribir programas.

Por lo tanto, en la teoría, podemos utilizar caracteres propios del lenguaje español para definir estructuras de datos, nombres de funciones, nombres de clases, etc.

Sin embargo, en la práctica, cuando desarrollamos un programa informático, no sabemos en qué entornos se acabará ejecutando. Desconocemos el hardware, el sistema operativo, el intérprete, y el idioma del entorno en el que nuestros usuarios ejecutarán nuestro programa. A lo largo de los años, en multitud de lenguajes de programación, he visto cómo esta cuestión puede ser un motivo por el cual nuestros programas funcionen en nuestro ordenador, y fallen al ejecutarse en los ordenadores, todos o algunos, de nuestros usuarios.

Por este motivo, no recomiendo utilizar caracteres especiales al nombrar variables, colecciones, y otras estructuras de datos, así como tampoco recomiendo ponerlos en las declaraciones de clases, y en definitiva, en cualquier identificador correspondiente a cualquier parte de un programa.

Sin embargo, sí que podremos poner perfectamente caracteres especiales y signos de puntuación en el interior de las cadenas, y en toda aquella parte de nuestro programa, que se va a mostrar al usuario final.

Así pues, como ejemplo, tendríamos el siguiente bloque de código:

```
informacion = "esta variable guarda información"
```

Donde, como podemos comprobar, en el propio identificador de la variable, he introducido una palabra en español que debería llevar acento, pero sin llevarlo, y dentro de la cadena alfanumérica de caracteres he puesto la palabra correctamente puntuado.

2.15. Cómo aprender a programar

Existe un dicho comúnmente extendido en el mundo de la programación y en el mundo específico de la docencia para la programación, que dice que " a programar, se aprende programando ".

Este dicho que puede parecer cruel en principio, sin embargo, refleja una realidad, y es que la única forma de aprender a programar es realizando programas.

Por lo tanto, en este punto de la publicación es donde debo parar un momento, y justo antes de entrar en lo que puramente podríamos considerar la materia principal de esta obra, voy a proponerte una forma de aprovechar los ejercicios que a continuación se te van a proponer.

Escribo este apartado de este capítulo, con el máximo respeto a la forma individual de aprender que tiene cada persona. No hay dos personas iguales, y esto al final se traduce en que cada persona tiene una forma diferente de aprender. A lo largo de estos años, y a través de muchos alumnos, he podido comprobar cómo al final la metodología docente, no tiene tanta importancia desde el punto de vista en el que no debe ser estática, sino adaptarse a la forma de aprender de cada alumno concreto, porque al final lo que importa es que el alumno aprenda, independientemente de la forma en la que lo haga.

A continuación, por tanto, te voy a proponer una metodología de aprendizaje, para que tú tomes mis palabras meramente como una propuesta y no como una imposición.

En las siguientes páginas y hasta el final del libro. Te voy a proponer una serie de ejercicios. El primer paso consiste en que copies y pegues

absolutamente todos los ejercicios que yo te voy a proponer en un bloc de notas, o bien en el editor integrado en Python, que, como hemos comentado anteriormente, recibe el nombre de Idle.

Una vez que hayas hecho eso, verás que, en los ejercicios, declaro una serie de variables o estructuras de datos utilizando unos identificadores. Bien pues, una vez que hayas Comprobado que mis ejercicios, tal cual yo te los planteo, funcionan correctamente, a continuación empieza a cambiar cosas del programa, cambian los nombres de las estructuras de datos, cambia los nombres de las cadenas, y, en definitiva, comprueba que poco a poco vas teniendo el control sobre la ejecución del programa, y que poco a poco puedes hacer cambios e ir adaptando el funcionamiento del programa a tu voluntad.

Cuando hayas conseguido eso, empieza a combinar y recombinar de nuevo las partes de los programas, copia y pega un trozo de un programa en otro programa, y comprueba que siguen funcionando, y así, poco a poco, copiando y pegando trozos, uniendo trozos, Darás un paso más hacia adelante en la construcción de tus propios programas.

Y solo cuando eso ocurra es cuando empezarás a estar preparado o preparada para empezar a crear tus propios programas desde cero. Pero en definitiva, en ningún caso esperes poder empezar a crear tus propios programas desde una página completamente blanco

Únicamente leyendo el texto que se encuentra dentro de este libro. Por el contrario, da por hecho que, como cualquier persona que aprende a programar, vas a empezar poco a poco, y vas a ir avanzando paso a paso, pero con paso firme.

2.16. El trabajo de programar

Otro de los paradigmas comúnmente aceptados, aunque sea a la fuerza y a regañadientes, por las personas que ya se dedican al mundo de la programación, es que, en contra de lo que parece, el trabajo de una persona que se dedica a la programación, la gran mayoría del tiempo, no consiste en crear programas, sino en buscar el origen de los errores que dan los programas.

Te adelanto, ya que durante el proceso de aprender a programar, en Python, al igual que en cualquier otro lenguaje, vas a cometer errores. Los vas a cometer tú, de la misma forma que los comete cualquier otra persona que está empezando a aprender a programar.

No se puede aprender a programar, sin haber cometido errores, y sin haber aprendido a encontrar el origen de esos errores, y solucionarlos. Lo bueno, en este caso, la suerte para ti en el caso de Python, es que Python es un lenguaje de programación bastante poco propenso a cometer errores, y por lo tanto te facilita el trabajo en este sentido.

Este apartado no trata la resolución de errores desde un punto de vista técnico, ya que eso lo haremos más adelante en el capítulo siguiente, sino que trata la resolución de errores de los programas informáticos desde el punto de vista de la frustración humana, desde el punto de vista de cómo una persona que se dedica a la programación, en un momento dado puede verse frustrada por la gran cantidad de errores con los que va a tener que tratar a lo largo del desarrollo de cualquier aplicación informática.

Simplemente, cuando esto ocurra, ten en cuenta que es algo completamente normal, le puedes preguntar perfectamente a cualquier persona que haya pasado anteriormente por este proceso, y de esa forma, probablemente, tu frustración será menos, y tu soledad, con suerte, también lo será.

2.17. Aspectos psicológicos del aprendizaje

Aprender a programar, si no se tiene la actitud correcta, puede ser duro. Para aprender a programar se requiere mucha más paciencia y constancia, que inteligencia. Así que si colocas tu actitud de forma correcta, podrás disfrutar de tu proceso de aprendizaje.

Lo primero que debes entender es que, al aprender a programar, tienes que aprender a pensar de la misma forma que lo hace una máquina - y la forma de pensar de las máquinas no se parece, en ocasiones, a la forma de pensar de las personas.

En primer lugar, las máquinas entienden dos estados: verdadero o falso. Y la mayoría de las ocasiones nuestros programas funcionan en base a estos dos

extremos. En el caso de las personas, prácticamente nada es completamente verdadero o completamente falso.

Las personas nos guiamos por el contexto, y aunque en el proceso de comunicación entre las personas haya palabras que no lleguemos a escuchar o que no entendamos completamente, las inferimos en base al contexto del momento. Las máquinas no infieren. Solo ejecutan al pié de la letra el código que les proporcionamos y, en muchas ocasiones, si en ese código falla un solo carácter, el programa no se ejecuta. Esto, al principio, es fuente de frustración - con el tiempo, sin embargo, simplemente se asume y se acepta.

Hay muchas personas que no pasan de ese proceso inicial de frustración, y abandonan la programación al no obtener la recompensa esperada. Bien, pues, si se tiene paciencia y constancia, después de esa frustración viene la recompensa.

"Aprender a andar antes de aprender a correr" es algo que repito en muchas ocasiones en las clases llenas de alumnos. El alumno tiene la visión romántica (visión fomentada por los audiovisuales que todos consumimos) de que, con poco esfuerzo, y en poco tiempo, podrá abrir una hoja en blanco y empezar a crear código desde cero. Ni funciona así, ni se espera que funcione así.

Primero se copia código de otras personas (para eso está este libro, para empezar). Mediante ese proceso de copia el alumno se familiariza con la sintaxis y con los errores. Luego, el código se modifica para personalizar su comportamiento. Más adelante se empieza a copiar, pegar y recombinar bloques de código, y durante todo ese proceso el alumno va, poco a poco y con práctica, interiorizando el funcionamiento de cada bloque y de cada instrucción. Y, con el tiempo, se tiene la suficiente fluidez como para poder escribir código desde cero. Ese momento llega, claro que sí, pero llega con la práctica y con el tiempo. No tengas prisa. Yo no la tengo. Solo preocúpate de ir poco a poco pero con paso firme.

Y no te pido que no te frustres, porque nadie ha aprendido a programar sin frustrarse y desesperarse en ocasiones, cuando parece que el código de vuelve rebelde. Cuando eso pase, aplica la técnica que mejor te funcione según tu forma de aprender: descansa un rato la mente, u obstínate en resolver el problema para el que no hallas solución, o en solucionar el error

del que no encuentras la causa. Pero, en todos los casos, no te tomes el proceso de aprendizaje de la programación como una carrera de velocidad, sino como una carrera de fondo.

3. Primeros pasos con Python

3.1. Entradas y salidas

3.1.1. Salidas

3.1.1.1. Impresión de información por pantalla - en la consola

Una de las primeras operaciones que vamos a realizar con Python, al igual que se realiza con cualquier lenguaje de programación, consiste en extraer información por pantalla, como mínimo utilizando la consola del sistema.

A lo largo de este libro vamos a comprobar cómo vamos a ser capaces de que nuestros programas traigan información mediante dos vías, en primer lugar, a través de la consola de ejecución, y en segundo lugar, a través del desarrollo de programas con interfaces gráficas, basadas en ventanas.

Sin embargo, en este caso, como estamos al principio del todo del libro, y por tanto, estamos al principio del todo de nuestro viaje a través de Python, vamos a empezar creando un sencillo programa que nos permita extraer una salida a través de la consola del sistema.

Para esto, lo que vamos a hacer es abrir un nuevo documento de Python, lo guardaremos con un nombre, el que queramos, y a continuación, pulsaremos sobre el comando que, en nuestro entorno de desarrollo, esté asociado a la ejecución del script.

El comando, por tanto, que introduciremos dentro de este primer programa, será el siguiente:

```
print("Hola mundo")
```

Al hacerlo, a través de la consola de ejecución, saldrá un sencillo mensaje que nos dirá " Hola mundo ", y que, por tanto, nos confirmará que la ejecución de nuestro primer programa ha sido exitosa.

```
Hola mundo
```

En desarrollo informático, los programas son entidades que cumplen tres funciones principales:

1. Admiten una entrada por parte del usuario
2. Realizan algún tipo de cálculo o procesamiento con la información aportada por el usuario
3. Extraen algún tipo de resultado por alguna de las vías que se consideren válidas.

Lo que hemos hecho, en este caso, es crear un pequeño primer programa que cumple la tercera de las características de un programa informático, y es que extrae información de alguna forma, en este caso presentándola en pantalla para que el usuario pueda tener algún tipo de retroalimentación y pueda saber que el programa ha funcionado correctamente.

3.1.1.2. Impresión de múltiples líneas

En ocasiones, queremos que nuestros programas informáticos extraigan varias líneas a través de la consola durante el proceso de ejecución. Para esto, podemos repetir el comando de impresión en pantalla, el comando print, tantas veces como necesitemos, con el objetivo de extraer y representar en la pantalla todo aquello que el programa nos quiera comunicar.

En este sencillo ejemplo que muestro a continuación, tenemos un nuevo programa que genera tres líneas de salida de código a través de la consola.

```
print("Esto es una línea de texto")
print("Esto es otra línea")
print("Y esto es otra línea")
```

Al ejecutar el programa, podremos comprobar que el resultado en pantalla será el siguiente:

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
```

```
information.

========= RESTART:
/Users/josevicente/Desktop/Python/printmultilinea.py
========
Esto es una línea de texto
Esto es otra línea
Y esto es otra línea
```

3.1.2. Entradas

Con anterioridad, hemos comentado que la esencia de un programa informático consiste en que el usuario sea capaz de introducir información dentro del programa, el programa realice cálculos a partir de la información que ha introducido el usuario, y finalmente, devuelva una salida.

Hasta el momento hemos sido capaces de que nuestros programas devuelvan información en pantalla, mayormente a través de la consola hasta el momento, y también hemos empezado a ser capaces de conseguir que los programas realizan cálculos aritméticos.

A continuación, presentamos la función input de entrada, que va a permitir a nuestros programas que el usuario final puedan introducir la información que desee.

```
print("Dime tu nombre")
nombre = input()
#a continuación quiero ver si el sistema está guardando esa
información
print("Hola,",nombre)

print("Dime tu edad")
edad = input()
print("Tu nombre es",nombre,",Tu edad es de",edad,"años")
```

En el programa que se muestra a continuación, observamos cómo se crea una variable llamada nombre, y el valor de esa variable se asocia a la llamada a la función input.

Cuando el programa se ejecute, en lugar de finalizar la ejecución del programa, el cursor se quedará esperando la entrada del usuario. Es por esto que en la línea anterior, se ha introducido un print para informar al usuario de que es lo que se espera de él a continuación.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============ RESTART:
/Users/josevicente/Desktop/Python/entradas.py ============
Dime tu nombre
Jose Vicente
Hola, Jose Vicente
Dime tu edad
44
Tu nombre es Jose Vicente ,Tu edad es de 44 años
```

A partir de ese momento, si hacemos un *Print*, podremos obtener por pantalla el valor de la variable que previamente el usuario ha introducido.

En la siguiente parte del programa vemos demostrado, como no solo estamos limitados a que el usuario introduzca una pieza de información, sino que podemos realizar tantas preguntas como necesitemos, para que el usuario introduzca tanta información como sea necesaria.

Vemos, por tanto, cómo podemos preguntarle al usuario, no solo su nombre, sino también su edad, para, a continuación, imprimirla en la pantalla utilizando, para este ejemplo, un encadenamiento con el signo de la, en forma de tu plan

3.2. Gestión de errores en Python

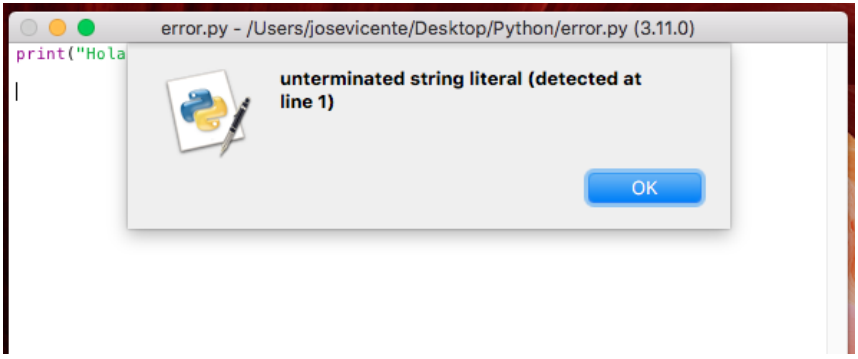
Crear programas informáticos sin cometer errores durante la escritura de los mismos, es un objetivo prácticamente imposible e inviable.

Debemos ser conscientes de que vamos a cometer dichos errores, y por lo tanto debemos anticipar cuál es van a ser, en la medida de lo posible, dichos errores, para poder arreglarlos cuando aparezcan y cuando sea necesario.

```
print("Hola mundo)
```

en el ejemplo anterior, si nos fijamos, al establecer un sencillo comando como el que ya conocemos, que es un comando print que saca información por pantalla, veremos que dentro de los paréntesis, la cadena de texto alfanumérico que he introducido, tiene una comilla, B inicio, pero he olvidado poner la comilla de finalización.

Si ejecuto este código, lanzará un error en la pantalla, que nos dirá que hay una instrucción mal terminada.



A continuación vamos a analizar a grandes rasgos, qué tipos de errores nos podemos encontrar en Python, y cómo podemos solucionar la gran mayoría de ellos de una forma bastante sencilla.

3.2.1. Errores sintácticos

En primer lugar nos encontramos los errores sintácticos.

Éstos son aquellos errores en los cuales el usuario ha cometido algún tipo de error al escribir el código, y ha roto alguna regla relacionada con la sintaxis del lenguaje de programación.

Existen una gran cantidad de errores sintácticos que podemos cometer, a continuación indico los más frecuentes para que podamos arreglarlos llegado el momento.

3.2.1.1. Errores de comillas

Como hemos visto en el caso anterior, un error muy frecuente, consiste en no cerrar correctamente las comillas.

Muchas veces nos acordamos de abrirlos correctamente, pero olvidamos completamente cerrarlas una vez que hemos finalizado de escribir la cadena alfanumérica de texto.

Por lo tanto, deberemos tener especial cuidado de abrir y cerrar Las comillas que escribamos en nuestro código.

3.2.1.2. Errores de paréntesis

Ocurre algo similar con los paréntesis que, sin duda, necesitaremos a lo largo de la escritura de nuestro código.

Para ello, deberemos comprobar que todos los paréntesis que hayan sido abiertos, queden cerrados en algún momento o en otro dentro de nuestro código.

3.2.1.3. Errores de encadenamiento

Un error muy típico y característico de cuando escribimos código consiste en olvidar realizar encadenamientos entre cadenas de caracteres y variables. Deberemos recordar que en Python existen varias formas de encadenar elementos, tales como, por ejemplo, el signo de adición y la coma, pero en

todo caso, siempre debemos utilizar uno de estos caracteres cuando encadenamos diferentes elementos.

3.2.2. Errores lógicos

Existe otra categoría de errores, que no consiste en cometer errores de sintaxis, sino en cometer errores de tipo lógico. Mediante estos errores, introducimos una condición dentro de la ejecución del programa, que no tiene solución, o que lleva al programa informático a cometer un error.

Estos son los errores más difíciles de solucionar, puesto que no arrojan un error como tal dentro de la sintaxis del código fuente de nuestro programa, sino que arrojan el error, cuando el intérprete intenta comPilar y ejecutar, paso a paso la aplicación que hemos desarrollado.

3.3. Comentarios

3.3.1. Introducción a los comentarios

Los comentarios consisten en una serie de entidades que encontramos en la gran mayoría de lenguajes de programación, que nos permiten dejar anotaciones en lenguaje completamente humano, donde la máquina ignorará completamente estas anotaciones en el momento de realizar la ejecución del programa.

Los comentarios cumplen varias funciones, entre las cuales podemos destacar dos principales.

La primera de ellas consiste en dejarnos anotaciones para nosotros mismos, para recordar, a nuestro yo futuro, que es lo que hacía cada una de las líneas de nuestro código, o qué es lo que hacía un bloque de código concreto dentro de nuestro programa informático.

La segunda de estas funciones consiste en dejar anotaciones preparadas para que otras personas puedan aprovecharlas, en el caso de trabajar en equipo, su trabajo, o en el caso de que varias personas durante un periodo

de tiempo, presumiblemente largo, puedan acceder y editar el código que estamos desarrollando.

De la misma forma, evidentemente nosotros mismos nos podremos beneficiar de los comentarios que otros desarrolladores hayan podido dejar en un código fuente que, en un momento dado, nosotros heredemos y sobre el cual tengamos que trabajar.

3.3.2. Comentarios

Los comentarios más sencillos con los que podemos trabajar son aquellos que consisten en una única línea de comentario. En Python se introducen introduciendo el carácter de la #, comúnmente conocido hoy en día como #

A continuación de la introducción de ese carácter, podremos escribir todo el texto que necesitemos para dejar las anotaciones preparadas para poder interpretar el código fuente más adelante.

```
# Primero defino la edad
edad = 19
# Segundo imprimo por pantalla
print("Que sepas que tienes",(edad+5),"años")
# Ahora multiplico por dos
edad = edad * 2
# Y lo vuelvo a sacar por pantalla
print("Y ahora la edad es ",edad)
```

Si ejecutamos el código anterior, podemos confirmar que la salida de consola es la siguiente:

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
/Users/josevicente/Desktop/Python/comentarios.py ==========
```

```
Que sepas que tienes 24 años
Y ahora la edad es  38
```

Y por tanto, apreciamos cómo en la ejecución del programa, los comentarios han sido completamente omitidos, y la aplicación únicamente lanza por consola aquellas líneas de código relacionadas con la declaración de la variable, la multiplicación, y la inclusión por pantalla

3.3.3. Comentarios multilínea

En ocasiones, cuando escribimos comentarios para anotar nuestro código fuente, es necesario que dichos comentarios abarquen varias líneas. Cuando esto ocurre, hay varias formas de realizar este tipo de comentarios.

Por supuesto una de ellas consiste en crear comentarios de una única línea, uno detrás del otro, tantas veces como necesitemos.

Sin embargo, existe otra forma que es razonablemente más cómoda, que consiste en poner tres comillas sencillas para empezar el comentario multilínea, y tres comillas sencillas para finalizarlo.

Todo el texto que se escriba dentro de estos bloques que acabo de describir, será considerado como un comentario, y por lo tanto será completamente ignorado por el intérprete en el momento de la ejecución de nuestro programa informático.

```
'''
Esta es la primera línea de comentario
Esta es la segunda línea de comentario
Esta es la tercera línea de comentario
'''
print("Esto es el programa en sí mismo")
```

Si ejecutamos el programa en la consola, podremos comprobar que el intérprete de Python ignorara, en la ejecución, todos y cada uno de los comentarios, y únicamente mostrará el resultado de la interpretación de la línea correspondiente a la operación de impresión.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

====== RESTART:
/Users/josevicente/Desktop/Python/comentariosmultilinea.py
=====
Esto es el programa en sí mismo
```

3.3.4. Comentarios por bloque y comentarios por línea

Existen muchas formas de introducir comentarios en el código fuente de una aplicación informática. Dos estilos de comentarios que podemos utilizar son los comentarios por línea, y los comentarios por bloque. Los comentarios por línea, intercalados o situados a la derecha con sangrado, nos informan del cometido de cada una de las instrucciones, y los comentarios por bloque nos informan del sentido de un bloque de código completo.

En el siguiente ejemplo podemos ver la utilización de comentarios por bloque y comentarios por línea, intercalados:

```
### en primer lugar creo variables ######################

# Creo una variable edad y le asigno el valor 44
edad = 44
# creo una variable multiplicador
multiplicador = 2

##### en segundo lugar realizo cálculos #################
# creo una nueva variable llamada "doble" y le asigno una
operación aritmética
doble = edad * multiplicador
```

```
##### por último imprimo resultados por pantalla #######

# imprimo un mensaje con la operación que voy a realizar
print("A continuación imprimo el doble de mi edad")
# imprimo mi edad actual
print("si mi edad es de"+str(edad))
#imprimo el doble de mi edad
print("el doble de mi edad es de "+str(doble)+" años")
```

3.3.5. Profusión de comentarios

El ejemplo anterior puede parecer una muestra de código en el que existe una sobrecarga de comentarios. Siendo esto cierto, no hay una sola forma de escribir código fuente. Especialmente en Python, que es un lenguaje donde se fomenta la legibilidad del código, se alienta a los programadores a escribir tantos comentarios como sea necesario, para que el código resulte legible.

Aún así hay que ser conscientes de que existen disciplinas, tales como por ejemplo la de la refactorización, que indican que, en ciertas circunstancias, si por ejemplo hace falta un comentario para definir lo que guarda una variable, es que probablemente sería necesario quitar el comentario y renombrar la variable para que su identificador resulte más intuitivo.

En todo caso, un estudiante de Python no escribe código de la misma forma que un experto lo hace (aunque en la teoría debería ser el mismo código): las personas que arrancan su camino en la programación tienden a poner muchos comentarios, o bien para entender el código, o bien para documentarlo para que sea revisado por compañeros o profesores. Una persona experta en Python probablemente omitirá muchos comentarios puesto que no serán necesarios para comprender lo que hace el código.

3.3.6. La estructura de un programa

El ejemplo anterior también ilustra otro concepto importante, y no necesariamente relacionado de forma estricta con los comentarios, que es la estructura que tiene un programa.

A lo largo del tiempo se ha intentado crear una estructura previsible y repetible para los programas informáticos: la realidad es que cada programa informático acaba siendo muy diferente de cualquier otro, y por lo tanto no es viable crear unas reglas demasiado estrictas a este respecto.

Dicho esto, sí que hay, al menos para los programas más básicos, una regla acerca de cómo estructurar el contenido de un programa, que puede ser de ayuda a las personas que empiezan a escribir código.

Por lo tanto, en un programa informático:

1. Se declaran los datos de entrada
2. Se realizan cálculos con los datos de entrada
3. Se representan los resultados en la salida

Por supuesto, este esquema se va haciendo más complejo conforme vamos necesitando recursos más complejos. Por ejemplo, en el caso de usar librerías, el esquema sería:

1. Se importan las librerías necesarias
2. Se declaran los datos de entrada
3. Se realizan cálculos con los datos de entrada
4. Se representan los resultados en la salida

Y si nuestro programa requiere usar funciones y/ clases, y dichas funciones y dichas clases no están situadas en librerías externas, el esquema se convierte en:

1. Se importan las librerías necesarias
2. Se declaran las clases necesarias
3. Se declaran las funciones necesarias
4. Se declaran los datos de entrada
5. Se realizan cálculos con los datos de entrada
6. Se representan los resultados en la salida

4. Estructuras de datos

4.1. Introducción a las estructuras de datos

Detrás de este nombre, tampoco atractivo, se esconde el concepto de que los programas informáticos, en mayor o menor medida, todos ellos tienen la necesidad de almacenar información en diferentes tipos de contenedores, dependiendo de la naturaleza de cuál sea esa información.

Las estructuras de datos, por lo tanto, son esos contenedores de información que vamos a necesitar durante el desarrollo de nuestros programas para guardar de forma temporal la información, y poder utilizarla en aquellas partes del programa donde sea necesario.

4.2. Variables

Las variables son la estructura de datos más básica que podemos encontrar en Python y en prácticamente cualquier lenguaje de programación para poder guardar información para más adelante.

Un sencillo programa que demuestra el uso de variables es el que se muestra a continuación.

```
edad = 19

print("Que sepas que tienes",edad,"años")
```

En este programa, en primer lugar, se declara una variable llamada edad, a la cual se le asigna el número 19.

A continuación, sacamos por pantalla el valor de la variable, encadenando con un texto, que en este caso recibe el nombre de String o cadena.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.
```

```
============ RESTART:
/Users/josevicente/Desktop/Python/variables.py ===========
Que sepas que tienes 19 años
```

Es muy importante que nos demos cuenta de que para encadenar un texto y una variable, especialmente cuando se trata de una variable numérica, estoy usando las comas como carácter separador.

Debemos tener cuidado de poner estas comas, ya que si no la ejecución del programa fallará.

4.2.1. Tipos de datos

Dentro de los diferentes tipos de lenguajes de programación que existen actualmente, y dentro de las múltiples formas de clasificar a estos lenguajes de programación, uno de los criterios de clasificación se basa en la tipificación de las variables.

La tipificación de las variables hace referencia a la precisión con la cual podemos especificar el tipo de datos que van a contener dichas variables.

Existen lenguajes de programación donde el programador está obligado a especificar cuál es este tipo de datos para cada una de las variables que va a utilizar, y existen otros lenguajes de programación, de nivel superior, como es, por ejemplo el caso de Python, llamados “débilmente tipificados”, en los cuales el usuario no declara el tipo de variable que va a utilizar, y es el intérprete, el que le asigna un tipo de forma dinámica durante la ejecución del script.

4.2.2. Forzado de tipo

Los lenguajes débilmente tipificados, como Python, están pensados para ser más sencillos de usar por parte del usuario final, ya que le quitan al usuario la responsabilidad de tener que utilizar los tipos correctos de datos, y cargan esta responsabilidad sobre el intérprete.

Sin embargo, en ocasiones ocurre que el intérprete puede equivocarse, como prácticamente siempre ocurre cuando dejamos que una máquina tome las decisiones.

Es por esto que debemos ser conscientes de que, aunque Python sea un lenguaje débilmente tipificado, es posible forzar el tipo de datos que en un momento dado usamos dentro de cualquiera de nuestros programas.

Para ilustrar este ejemplo, pongo el programa a continuación.

```
print("Dime la edad")
edad = input()
print("El doble de tu edad es",(edad+2))
```

En este caso, estamos haciendo uso de la función de entrada al sistema de información, que es la función input, que, en todo caso, siempre devuelve la información como una variable de tipo String, es decir, una variable de tipo cadena de datos, alfanuméricos, o lo que es lo mismo, mayormente Cadenas de texto.

Pero, sin embargo, si nos fijamos, para este ejemplo, en el cual necesito calcular el doble de una edad, necesito que esa variable sea interpretada como número entero y no como cadena de texto, ya que si no lo hago de esa forma, la operación aritmética que quiero realizar a continuación, podría fallar.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
/Users/josevicente/Desktop/Python/cambiodetipo1.py =========
Dime la edad
44
Traceback (most recent call last):
  File "/Users/josevicente/Desktop/Python/cambiodetipo1.py",
```

```
line 3, in <module>
    print("El doble de tu edad es",(edad+2))
TypeError: can only concatenate str (not "int") to str
```

Es por esto que se ha utilizado, en el ejemplo anterior, la función *int*, es decir, una función que nos permite convertir cadenas de texto en números enteros, para que a continuación pueda realizar operaciones aritméticas sin ningún tipo de problema sobre el dato de entrada.

```
print("Dime la edad")
edad = int(input())
print("El doble de tu edad es",(edad+2))
```

Si ejecutamos el código anterior podremos comprobar cómo a partir de una edad original, introducida a partir de una función *input*, obtendremos el doble de la edad introducida, habiendo usado la función de conversión a números enteros.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
/Users/josevicente/Desktop/Python/cambiodetipo2.py =========
Dime la edad
44
El doble de tu edad es 88
```

4.3. Listas

4.3.1. Definición de una lista

En una gran cantidad de ocasiones, las variables, en un momento dado, se quedan cortas en cuanto a capacidad de almacenamiento, cuando, dentro de ese espacio, no necesitamos introducir una sola pieza de información, sino que necesitamos que el sistema guarde un listado de datos.

Existen una gran cantidad de estructuras de datos que podemos utilizar dentro de Python, pero sin duda la preferida es una lista.

Mediante las listas, podemos guardar, dentro de un contenedor de información, diferentes Dos, introduciendo estos elementos dentro de una lista encerrada por corchetes, indicada por “, y separados por comas.

```
agenda = ["Jose Vicente","Juan","Jorge","Jose"]

print(agenda[3])
```

En el ejemplo anterior, declaramos una agenda, y en lugar de introducir un solo elemento, a continuación especificamos una lista de nombres que va a contener esa agenda.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============= RESTART:
/Users/josevicente/Desktop/Python/lista1.py =============
Jose
```

En la segunda parte del ejemplo, mostramos cómo llamar a uno de los elementos de esa lista, teniendo en cuenta que los elementos quedan automáticamente guardados mediante un índice numérico que empieza en

el número cero. De esta forma, cuando llamamos al elemento número tres de la lista, realmente estamos llamando al cuarto elemento por orden

4.3.2. Impresión de la lista

```
agenda = ["Jose Vicente","Juan","Jorge","Jose"]

print(agenda)
```

Si ejecutamos este código, podremos comprobar, en la consola resultante, que, a diferencia de otros lenguajes de programación, y a diferencia de otros intérpretes, en Python obtenemos el volcado de la lista directamente en el terminal, lo cual resulta tremendamente conveniente para no tener que recorrer uno a uno los elementos de la lista para poder sacarlos en pantalla

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========= RESTART:
/Users/josevicente/Desktop/Python/impresionlista.py
=========
['Jose Vicente', 'Juan', 'Jorge', 'Jose']
```

4.3.3. Impresión de un elemento de la lista

Además de poder imprimir todos los elementos que contiene la lista, también vamos a poder especificar con cuál de los elementos queremos trabajar.en el siguiente ejemplo, trabajamos con el cuarto elemento, representado por el índice número 3, ya que debemos tener en cuenta y recordar que en las listas, al igual que en la mayoría de estructuras de datos en los lenguajes de programación, el índice inicial es el número 0

```
agenda = ["Jose Vicente","Juan","Jorge","Jose"]

print(agenda[3])
```

De la misma forma que anteriormente hemos volcado en la terminal toda la lista completa, comprobamos que ahora aparece en la consola únicamente el elemento que hemos seleccionado.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

===== RESTART:
/Users/josevicente/Desktop/Python/impresionelementolista.py
=====
Jose
```

4.3.4. Listas multidimensionales

El ejemplo de la agenda es muy común cuando se enseña a trabajar con estructuras de datos tales como listas, y sirve precisamente para forzar el poder mostrar. A continuación la utilización de listas multidimensionales.

Si lo que estamos creando es una agenda, generalmente la agenda no suele contener sólo nombres, sino que suele contener también otros elementos, tales como por ejemplo, teléfonos y correos electrónicos.

De esta forma, por tanto, no solo podemos realizar una lista de una sola dimensión como aquella que hemos realizado en el ejercicio anterior, sino que podemos anidar unas listas de otras, para crear listas multidimensionales.

En el siguiente ejercicio se presenta un ejemplo en el cual creamos una vista bidimensional, en la cual, en primer lugar, creamos una lista, y luego, dentro de cada uno de los elementos dentro de esa agenda, introducimos otra lista.

```
agenda = list(range(100))
agenda[0] =
["JoseVicente","5453425","info@josevicentecarratala.com"]
agenda[1] = ["Juan","111111","juan@josevicentecarratala.com"]

print(agenda[0][1])
```

Al ejecutar este programa podremos comprobar como en la consola aparece únicamente el dato que hemos solicitado, que para el ejemplo anterior consiste en seleccionar el primer elemento de la primera dimensión de la lista, y dentro de ese primer elemento, seleccionamos el segundo elemento, que en este caso corresponde al teléfono.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

===== RESTART:
/Users/josevicente/Desktop/Python/listamultidimensional1.py
=====
5453425
```

Al igual que ocurre con las listas de una única dimensión, si en un momento dado deseamos imprimir toda la lista, simplemente podemos obtenerla en la terminal introduciendo el comando *PRINT*.

```
agenda = list(range(100))
agenda[0] =
["JoseVicente","5453425","info@josevicentecarratala.com"]
agenda[1] = ["Juan","111111","juan@josevicentecarratala.com"]

print(agenda)
```

Al realizar la impresión comprobamos como la lista completa aparece en la consola, independientemente del número de dimensiones que contenga.

Esto puede ser tremendamente útil para aquellos casos en los que estemos utilizando la consola para localizar errores, para *debuggear*, o simplemente para comprobar cuál es el estado actual de ejecución del programa.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

===== RESTART:
/Users/josevicente/Desktop/Python/listamultidimensional2.py
=====
[['JoseVicente', '5453425', 'info@josevicentecarratala.com'],
['Juan', '111111', 'juan@josevicentecarratala.com'], 2, 3, 4,
5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65,
66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
96, 97, 98, 99]
```

4.3.5. Estructuras mutables e inmutables

Algunas estructuras de datos presentan la característica de tener mutabilidad, y otras, como las tuplas, son inmutables.

Esto quiere decir que cuando declaramos una lista, en primer lugar, tenemos que declarar cuál va a ser la longitud

4.3.6. Operaciones con Listas

Una vez que trabajamos con cualquier estructura de datos, en este caso las listas, debemos conocer las operaciones que podemos realizar con las mismas.

En primer lugar, dentro del intérprete, por defecto de Python, podremos comprobar que, en un momento dado, podemos lanzar a la consola una impresión de una lista.

Encontraremos tremendamente cómodo comprobar que cuando pedimos imprimir una lista por consola, sale el listado de todos los elementos de esa lista, dentro de la consola, lo que resulta altamente conveniente cuando estamos dibujando un programa.

Otra característica interesante de las listas en Python es que podemos recorrerlas en sentido negativo. De esta forma, si llamamos al elemento que está en el índice número cero de la lista, estaremos llamando al primero de los elementos contenidos, pero si por ejemplo llamamos al elemento con el índice -1, encontraremos que estaremos llamando al último elemento de la lista.

4.3.6.1. Definición de una lista

A continuación, vamos a desarrollar una serie de ejercicios para comprobar que trabajamos correctamente con listas y con las operaciones más comunes que podemos realizar con ellos.

En primer lugar vamos a realizar una definición de una lista de tres nombres propios de personas, de la siguiente forma

```
nombres = ['Jose','Juan','Jorge']
```

4.3.6.2. Impresión de la lista

Una de las primeras operaciones que podemos realizar con la lista que hemos definido previamente, consiste en imprimir la lista por pantalla. Por lo tanto, en primer lugar, a continuación del comando que hemos utilizado anteriormente para declarar una lista, simplemente imprimimos la lista por pantalla

```
nombres = ['Jose','Juan','Jorge']
print(nombres)
```

4.3.6.3. Acceso a un elemento de la lista

Si en el ejercicio anterior hemos impreso la lista completa, veremos a continuación que podemos imprimir un elemento concreto de la lista, simplemente llamando a su índice.

Para el cálculo de los índices. Deberemos tener en cuenta que las estructuras de datos en Python, como las estructuras de datos en prácticamente cualquier lenguaje de programación empiezan por el índice numérico cero en lugar de empezar por el índice numérico uno.

```
nombres = ['Jose','Juan','Jorge']
print(nombres)

print(nombres[1])
```

4.3.6.4. Acceso a elementos con índice negativo

Una característica bastante interesante de Python y que no se encuentra en otros lenguajes de programación es que si introducimos un número negativo

en el índice de la lista al cual queremos acceder, el sistema llama a los últimos elementos de la lista.

Por lo tanto, si llamamos al elemento -1 de la lista, obtendremos el último elemento, si llamamos al elemento -2 de la lista, obtendremos el penúltimo elemento, y así sucesivamente

```
nombres = ['Jose','Juan','Jorge']
print(nombres)

print(nombres[1])

print(nombres[-1])
```

4.3.6.5. Añadido de nuevos elementos a la lista

Al contrario, que ocurre en otros lenguajes de programación, todavía menos estrictos que Python, en los cuales podemos asignar de forma dinámica nuevos elementos dentro de una colección de datos, simplemente llamando al índice con un nuevo número, en Python tenemos que utilizar la instrucción Apple para añadir un nuevo elemento al final de la lista.

En el siguiente ejercicio se puede ver ilustrado el método mediante el cual añadimos un nuevo elemento al final de nuestra lista.

```
nombres = ['Jose','Juan','Jorge']
print(nombres)

print(nombres[1])

print(nombres[-1])

# Mutables
```

```
# nombres[3] = 'Julia'
nombres.append('Julia')
```

4.3.6.6. Eliminación de elementos de la lista

Siguiendo con la filosofía y la metodología que estábamos trabajando en el ejercicio anterior, no es posible eliminar un elemento de la lista sin utilizar un comando específico, que en este caso será el comando pop.

Esta instrucción nos permite eliminar un elemento concreto de la matriz, especificando, dentro de los paréntesis que requiere la llamada a este método, el índice numérico del elemento que queremos eliminar.

```
nombres = ['Jose','Juan','Jorge']
print(nombres)

print(nombres[1])

print(nombres[-1])

# Mutables

# nombres[3] = 'Julia'
nombres.append('Julia')
print(nombres)
nombres.pop(0)
print(nombres)
```

4.3.6.7. Sobre escritura de elementos de la lista

Una vez que hemos definido unos valores iniciales para la colección, deberemos ser conscientes de que cada uno de los elementos de la lista se

comporta como una variable, en el sentido de que aloja una pieza de información en una dirección de la memoria, y esa información se puede sobre escribir en cualquier momento, simplemente llamando al índice de la matriz, y estableciendo un nuevo valor, de la misma forma que lo haríamos en el caso de sobre escribir el valor de una variable.

```
nombres = ['Jose','Juan','Jorge']
print(nombres)

print(nombres[1])

print(nombres[-1])

# Mutables

# nombres[3] = 'Julia'
nombres.append('Julia')
print(nombres)
nombres.pop(0)
print(nombres)
nombres[0] = 'Julia'
```

Las listas, como hemos comentado anteriormente, son estructuras de datos mutables.

Por tanto, esto quiere decir que en un momento dado podemos añadir y podemos quitar tantos elementos como sea necesario. Como podemos ver en el ejemplo anterior, hemos utilizado la instrucción Append para introducir un nuevo elemento al final de la lista, y podemos utilizar también la instrucción pop para eliminar el elemento de la lista que queramos.

Por último, de la misma forma que podemos comprobar que es necesario utilizar instrucciones tales como Apple o pop para añadir o quitar elementos de la lista, recordaremos que para cambiar un elemento de la lista, simplemente tenemos que asignar de nuevo el valor de su índice.

4.4. Tuplas

Mientras que las listas son estructuras de datos mutables, encontramos en las tuplas una estructura de datos inmutable, que, tal y como hemos analizado anteriormente, nos puede servir para aquellos casos en los que necesitemos un rendimiento superior en nuestra aplicación.

Las tuplas, por tanto, son estructuras de datos inmutables que, una vez definidas en longitud, no pueden cambiar.

Podría parecer que es un caso inconveniente por regla general dentro del mundo de la programación, pero sin embargo es más común de lo que en un principio podría parecer.

Por ejemplo, si dentro de una tupla, estamos metiendo las tres dimensiones que definen la posición de un objeto, X, Y, y Z, podremos ver fácilmente que no Una cuarta dimensión, o al menos si existe, no es perceptible para el ser humano, por lo tanto, no sería necesario aumentar, a mitad de un programa, la cantidad de dimensiones del espacio.

Por ejemplo, en el caso de la agenda que tratamos anteriormente, Una vez que se define el modelo de datos de una aplicación, en este caso decidimos que una agenda guarda el nombre, el teléfono, y un correo electrónico, no es común, o no es frecuente cambiar ese modelo de datos a mitad del desarrollo de una aplicación.

Existen, por tanto, múltiples ejemplos, en los cuales las listas son convenientes por su mutabilidad, y existen, por supuesto, también, múltiples ejemplos en los cuales las tuplas son igualmente convenientes precisamente por lo contrario que las listas.

4.5. Definición de una tupla

Y en el siguiente ejemplo, por tanto, vamos a crear una tupla, y a continuación, simplemente por una cuestión de comparación, vamos a crear una lista.

Veremos que la diferencia entre ambos tipos de declaración consiste en que la información contenida en una tabla se marca entre paréntesis, mientras que la información contenida en una lista se enmarca entre corchetes.

```
##Lista - Mutables - con corchetes
##Tuplas - Inmutables - paréntesis


longaniza = ("uno","dos","tres","a")
morcilla = ["uno","dos","tres","a"]


morcilla[3] = "cuatro"

print(longaniza[1])
print(morcilla[1])

print(len(longaniza))
print(len(morcilla))
```

Comprobaremos que podemos utilizar métodos muy parecidos para acceder a la información con respecto a los que hemos utilizado en las listas.

Eso sí, mientras que en una lista encontramos la mutabilidad, en el sentido de que podemos introducir nuevos elementos de la lista cuando sea necesario, deberemos tener en cuenta siempre de que en una dupla no podemos realizar esa operación.

4.6. Impresión de una tupla

A continuación vamos a imprimir elementos de una dupla, simplemente para comprobar que los métodos de impresión son exactamente iguales que los de las listas.

En este punto es importante ver que, mientras que la dupla se declara con paréntesis para diferenciar la verdad declaración de la lista, sin embargo, en el momento en el que queremos llamar a un elemento de la tabla, deberemos llamarlo con corchetes, de la misma forma que llamaríamos a un elemento de la lista.

```
##Lista - Mutables - con corchetes
##Tuplas - Inmutables - paréntesis


longaniza = ("uno","dos","tres","a")
morcilla = ["uno","dos","tres","a"]


morcilla[3] = "cuatro"

print(longaniza[1])
print(morcilla[1])

print(len(longaniza))
print(len(morcilla))
```

Igualmente las tuplas son inmutables, pero sin embargo eso no quiere decir que sus valores sean constantes, sino que son variables y los podemos modificar. En el ejemplo anterior, vemos cómo podemos cambiar el elemento que se encuentra en un límite concreto, simplemente utilizando el operador de asignación, el signo igual

4.7. Impresión de la longitud de la tupla

Otro de los métodos que podemos utilizar en las tuplas, de la misma forma que lo podemos utilizar en las listas es la longitud. Mediante la longitud, en este caso, podemos averiguar cuál es la cantidad de elementos que contiene una tabla.

```
##Lista - Mutables - con corchetes
```

```
##Tuplas - Inmutables - paréntesis


longaniza = ("uno","dos","tres","a")
morcilla = ["uno","dos","tres","a"]


morcilla[3] = "cuatro"

print(longaniza[1])
print(morcilla[1])

print(len(longaniza))
print(len(morcilla))
```

4.8. Tuplas

Las tuplas son estructuras de datos que, en principio, presentan inmutabilidad.

Aún así, en ocasiones, puede ser necesario ampliar el tamaño de una tupla, para lo cual presentamos un sencillo truco que consiste en utilizar las técnicas de conversión de tipo que podemos encontrar en el lenguaje de programación Python.

En este caso, sabemos que las tablas son inmutables, pero también sabemos que las listas son mutables.

La técnica que presentamos a continuación consiste en convertir una tabla en una lista, para a continuación, realizar una operación de añadido de un nuevo elemento al final de la lista, y por último, volver a convertir la lista en una tupla, para poder seguir beneficiándonos de las ventajas que nos proporciona esta estructura de control.

```
nombres = ('Jose','Juan','Jorge')
```

```
print(nombres)
print(nombres[0])

##nombres[0] = 'Julia'

##nombres.append('Julia')

#truco

lnombres = list(nombres)
lnombres.append('Julia')
nombres = tuple(lnombres)
print(nombres)
```

4.9. Conjuntos

Los conjuntos en Python se declaran de forma similar a los diccionarios, en el sentido de que usamos llaves para definirlos, aunque internamente se separan con comas, como las listas o las tuplas.

Los conjuntos pueden contener múltiples tipos de datos (pueden combinar strings y números, por ejemplo), pero no pueden contener otras estructuras de datos, como por ejemplo listas.

Algo muy importante con respecto a los conjuntos, es que, internamente, no funcionan como colecciones, en el sentido de tener un índice numérico, sino que su índice se calcula mediante un hasheado del valor que guarda la estructura de datos. Esto quiere decir que los valores de un conjunto son únicos (dos valores idénticos proporcionan el mismo hash). Por lo tanto son ideales en aquellos escenarios en los que necesitamos que los elementos de una estructura de datos no se puedan repetir.

```
nombres = {'Jose','Juan','Jorge'}
```

```
print(nombres)

for x in nombres:
    print(x)

print('Jose' in nombres)
nombres.add('Julia')
print(nombres)

nombres.remove('Julia')
print(nombres)
```

4.10. Diccionarios

Los diccionarios son otra de las estructuras de datos que encontramos dentro del lenguaje Python. A diferencia de otras estructuras de datos, nos permiten guardar parejas de atributo y valor, donde el atributo puede tener un nombre, lo cual las hace muy convenientes para guardar información consistente en nombres de #, y valores de #.

Por otra parte, como podemos observar en el ejemplo siguiente, en el cual, en primer lugar, declaramos un diccionario, su sintaxis es altamente similar a la sintaxis que tiene Los archivos de tipo J son, lo cual lo hace especialmente conveniente para introducir información dentro del sistema en este formato, y también para realizar el camino inverso.

```
nombres = {
        "accesible":"Que tiene acceso",
        "aacesit":"Segundo premio en un concurso cientifico",
        "2":"Jorge",
        "3":"Javier",
        "4":"Julia",
    }
```

```
print(nombres)

print(nombres['accesible'])

for i in nombres:
    print(nombres[i])
```

En la segunda parte de este ejemplo podemos comprobar cómo, al igual que ocurre con otras estructuras de datos, podemos imprimir directamente un diccionario en pantalla.

Por supuesto, también es perfectamente posible imprimir uno de los elementos del diccionario, simplemente accediendo al nombre de su clave, de la misma forma que lo haríamos accediendo al elemento de una lista.

Y luego, por último, podemos comprobar cómo podemos recorrer fácilmente un diccionario utilizando una sencilla estructura *for*, mediante la cual convertimos los datos de un diccionario a un número de un iterador, y a partir de ese momento podemos iterar en el diccionario sin ningún tipo de problema.

5. Operadores

5.1. Operador de encadenamiento

El primer operador que debemos aprender es el de encadenamiento, que nos permite encadenar diferentes operadores, especialmente cuando se tratan de operandos basados en cadenas.

En Python, el operador de encadenamiento es el signo +, lo cual en un primer momento puede causar confusión, porque, como veremos a continuación, en el siguiente apartado, ese mismo operador, se utiliza como operador aritmético de suma.

A continuación se muestra un ejemplo mediante el cual podemos comprobar cuál es el uso de operador de encadenamiento, por ejemplo para encadenar una serie de cadenas de caracteres.

5.2. Operadores aritméticos

Los operadores aritméticos son los más sencillos de entender, ya que representan las operaciones más básicas de uso frecuente y cotidiano, y por tanto no son ampliamente familiares.

En el siguiente ejemplo, podemos comprobar el uso de operaciones aritméticas básicas, tales como por ejemplo, la suma y la multiplicación.

```
edad = 19

print("Que sepas que tienes",(edad+5),"años")

edad = edad * 2

print("Y ahora la edad es ",edad)
```

Al ejecutar el código anterior, podemos comprobar el resultado en la consola, qué consiste en que en primer lugar declaramos una variable, realizamos una suma sobre la variable, y lanzamos el resultado en la consola junto

A continuación, multiplicamos la variable por dos, pero de forma que estamos introduciendo el valor de forma permanente en la variable, para, a continuación, sacarla por pantalla.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

====== RESTART:
/Users/josevicente/Desktop/Python/operadoresaritmeticos.py
=====
Que sepas que tienes 24 años
Y ahora la edad es  38
```

Al igual que ocurre en la gran mayoría de lenguajes de programación, podremos encontrar que los operadores aritméticos básicos son:

- La suma, representada por el signo +
- La resta, representada por un signo -
- La multiplicación representada por un *
- La división, representada por una barra /
- El resto entero de la división, representado por el signo del porcentaje %

5.3. Órden de precedencia

En Python se aplican las mismas reglas de procedencia que podemos encontrar en el mundo matemático. De esta forma, cuando dentro de una expresión se encuentran diferentes operadores aritméticos, deberemos tener en cuenta que siempre la multiplicación y la división se van a resolver antes que la suma y la resta.

De todas formas, igualmente de la misma forma que encontramos en el mundo matemático, podemos hacer uso de los paréntesis para poder forzar

y especificar qué operaciones deseamos que se resuelvan primero, y qué operaciones deseamos que se resuelvan después.

De esta forma, siempre que el lenguaje de programación encuentre paréntesis e incluso paréntesis anidados, aplicará la regla de primero resolver los paréntesis que se encuentran más animados, y luego, poco a poco, ir abriendo el espectro hacia los paréntesis más superiores.

```
print(3 + (4 * 5))
```

Mediante el uso de los paréntesis, podemos especificar el orden en el cual deben resolverse las operaciones. En este ejemplo, comprobamos cómo, en primer lugar, se ejecuta el cálculo aritmético que encontramos dentro del grupo más interior de paréntesis, y a continuación realizamos el cálculo aritmético que se encuentra en el nivel superior.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
/Users/josevicente/Desktop/Python/precedencia.py ==========
23
```

5.4. Operadores lógicos

Los operadores lógicos nos permiten realizar evaluar expresiones que devolverán, al final, un resultado de verdadero o falso.

Existen multitud de operadores lógicos tales como por ejemplo la doble igualdad, que compara si el valor evaluado a la izquierda del operador es igual, es decir, tiene el mismo valor que el operador situado a la derecha del operador.

El operador de admiración, al igual que en otros muchos lenguajes de programación, es la negación. Por tanto, encadenar un ! y un signo de igualdad equivale a realizar la pregunta de si es cierto que el valor a la izquierda del operador no es igual que el valor a la derecha del operador.

```
print(3 == 4)

print(3 != 4)
```

Al ejecutar este programa en la consola, comprobamos que, para el primero de los ejemplos, el resultado es falso, ya que no es cierto que el número 3 sea igual a 4. En el segundo de los ejemplos que ilustra el bloque de código anterior, podemos comprobar que el resultado en la consola es verdadero, ya que efectivamente es cierto que el número 3 no es igual a 4.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

======== RESTART:
/Users/josevicente/Desktop/Python/operadoreslogicos.py
=======
False
True
```

5.5. Operadores booleanos

Los operadores booleanos, nos permiten evaluar conjuntos de expresiones, de la misma forma que podamos obtener un resultado global con respecto a esa expresión, ya sea verdadero, o sea falso.

Los dos operadores booleanos principales que encontramos son el operador *and* y el operador *or*.

Si vienen otros lenguajes de programación estos operadores bien representados por el *ampersand* (&) y por el símbolo de la barra vertical (|), en Python, deberemos introducir directamente las palabras reservadas *and* y *or*.

```
print(3 == 3 and 4 == 4)
print(3 == 3 and 4 == 5)

print(3 == 3 or 4 == 5)
```

Cuando ejecutamos el código anterior en la consola, podemos comprobar que, en el primero de los casos, la validación resulta ser cierta.

En el segundo de los casos, sin embargo, la validación resulta falsa, ya que la primera igualdad es verdadera, pero la segunda igualdad es falsa.

Sin embargo, en la tercera de las expresiones hemos utilizado el operador OR, con lo cual, dado que una de las expresiones validadas, o al menos una de las igualdades es cierta, el resultado global de la expresión completa da igualmente cierta.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

======= RESTART:
/Users/josevicente/Desktop/Python/operadoresbooleanos.py
======
True
False
True
```

En el caso del operador AND, para que el resultado global de la expresión resulte cierto, todos los operandos evaluados deben ser ciertos. En el caso del operador OR, para que el resultado global de la operación de cierto, solo basta con que uno de los operadores sea cierto.

6. Estructuras de control del flujo de la ejecución

6.1. Introducción

Las estructuras de control del flujo de la ejecución, también llamadas simplemente estructuras de control, nos permiten establecer el flujo de ejecución de un programa informático.

Dentro de la linealidad en la ejecución de una aplicación informática, en el sentido de que hay unos datos de partida, unos cálculos, que se realizan por parte del programa, y un resultado que se devuelve al usuario, Dentro de esos cálculos, a veces podemos tomar caminos condicionales, o podemos necesitar realizar operaciones repetitivas.

Es por esto que estas operaciones de control de flujo se pueden agrupar en dos grandes familias, que son las estructuras de control condicional, aquellas que ejecutan unas partes u otras del programa informático, en base a las evaluaciones previas, de unas condiciones, y las estructuras de ejecución de bucle, Que realizan operaciones de forma repetitiva y automática.

6.2. Estructuras condicionales

Dentro del primer grupo encontramos las estructuras condicionales. Con esas estructuras, se evalúa una expresión, y en base al resultado de la evaluación de expresión, se ejecuta una parte del código, o no se ejecuta. En ocasiones, se realiza la evaluación de esa expresión, y en el caso de que la evaluación resulte verdadera, se ejecuta una parte del código, y en el caso de que resulte falta, se ejecuta otra parte del código.

Las estructuras de control condicional están ampliamente ligadas a los operadores, ya que muchas de ellas parten de la evaluación de una expresión, que, finalmente, debe resultar en una validación de verdadero o falso.

6.2.1. If

La estructura de condicional *if*, o lo que es lo mismo, el “si condicional”, nos permite evaluar una expresión, y ejecutar una parte del código en el caso de que esa expresión sea cierta.

Deberemos tener en cuenta que esta estructura de control requiere que todo el código introducido dentro del caso cierto este sangrado. Cuando hablamos de sangrías, equivalen a introducir una tabulación al principio de la línea para que el texto esté alineado ligeramente a la derecha.

La estructura de control *if* tiene también un caso *else*, que es el caso negativo. Es decir, se evalúa una expresión, y en el caso de que esa expresión resulte cierta se ejecuta la primera parte del código, y en el caso de que la evaluación de esa expresión resulte falsa, se ejecuta todo el código que encontramos dentro del caso *else*.

Deberemos tener en cuenta todo el código contenido. Dentro del caso else, es decir, el caso falso, deberá estar igualmente sangrado, de la misma forma que lo está el código que encontramos dentro del caso *if*.

```
print("Dime tu nombre")
nombre = input()
#a continuación quiero ver si el sistema está guardando esa
información
print("Hola,",nombre)

print("Dime tu edad")
edad = input()
print("Tu nombre es",nombre,",Tu edad es de",edad,"años")

edad = 15

if edad < 30:
    print("Eres una persona joven")
else:
    print("Ya no eres tan joven como antes")
    print("Que sepas que esto está dentro del else")

print("esto se va a ejecutar sí o sí")
```

Al ejecutar el código podremos comprobar cómo podemos seleccionar el bloque de instrucciones que se ejecutará, en base a la validación del valor de la variable original, y podremos comprobar también igualmente como el

último bloque de código, es decir, la última instrucción de impresión, se va a imprimir de todas formas.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

=============== RESTART:
/Users/josevicente/Desktop/Python/if.py ===============
Dime tu nombre
Jose Vicente
Hola, Jose Vicente
Dime tu edad
44
Tu nombre es Jose Vicente ,Tu edad es de 44 años
Eres una persona joven
esto se va a ejecutar sí o sí
```

Al final de este ejemplo podemos comprobar cómo hemos introducido una instrucción *Print*, para demostrar que una vez que finaliza la evaluación del caso *if*, o de la estructura de control condicional *else*, el programa continúa su ejecución con total normalidad.

6.2.2. If anidado

Una de las reglas que podemos encontrar dentro del zen de Python, nos aconseja anidar el código lo mínimo posible. Sin embargo, en el siguiente ejemplo mostramos, para aquellos casos, donde es estrictamente necesario, un ejemplo de animación, de estructuras de control condicional, mediante el cual mostramos que es posible evaluar no solo un caso verdadero y falso, sino cuatro casos, anidando, sendas estructuras de control If, dentro de una estructura de control y principal.

```
edad = 25
if edad < 20:
    if edad < 10:
        print("Eres un niño")
    else:
        print("Eres un adolescente")
else:
    if edad < 30:
        print("Eres un joven")
    else:
        print("Ya no eres tan joven")
```

En este ejemplo podemos comprobar, en la ejecución en la terminal, que siendo que la era de entrada ha sido puesta en 25, el resultado que aparece en la consola corresponde a la cadena "eres un joven con mi nueva línea es decir, se ha entrado dentro de el primer bucle de validación, y dado que no es cierto que la edad sea menor que 20, se ha ejecutado el caso falso, caso falso en el cual encontramos una estructura de control anidada, que nos permite volver a realizar la verificación de si es cierto que la edad es menor que 30,o es falso.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============ RESTART:
/Users/josevicente/Desktop/Python/ifanidado.py ===========
Eres un joven
```

En este caso, podemos observar que en una primera evaluación condicional hemos preguntado acerca de si una edad es menor que 20, en cuyo caso encontramos un caso principal verdadero <20, y un caso falso, en el *else*, que quiere decir que la edad es mayor o igual que 20.

Dentro del caso verdadero, introducimos otra estructura de control condicional *if*, le preguntamos si además de ser menor que 20 también es menor que 10, en cuyo caso falso, encontraríamos una situación en la que la edad es menor que 20, pero mayor o igual que diez.

Lo mismo ocurre para el caso falso, en el cual hemos introducido una estructura de control condicional anidada, con lo cual podemos obtener, finalmente, un mayor control acerca de las evaluaciones que realizamos.

Las anidaciones de estructura de control siempre han tenido fama de ralentizar el rendimiento de los programas informáticos, y de constituir una mala práctica desde el punto de vista de la legibilidad y de la mantenibilidad del código.

Por esta razón, se recomienda, por regla general, reducir el uso de las anidaciones, las estructuras de control, ir al mínimo posible, de la misma forma que se reconoce que en muchas ocasiones es conveniente, o directamente inevitable, utilizar estas estructuras de control.

6.2.3. Else If

La estructura de control condicional *if* nos permite evaluar la posibilidad de ejecutar partes del código en base a una expresión que resulta verdadera o falsa.

Sin embargo, en ocasiones, la expresión evaluada no debe devolver verdadero o falso, sino que existen una serie de casos que pueden resultar verdaderos, y existen otra serie de casos que pueden resultar falsos.

Un ejemplo lo encontraríamos dentro de los días de la semana, si por ejemplo le realizamos la pregunta de “cuál es el día de la semana“ a un programa informático, en ese caso, la respuesta no sería verdadera o falsa, sino que habría siete respuestas verdaderas, y un número prácticamente infinito de respuestas falsas.

Para resolver este tipo de situaciones, muy comunes dentro del mundo de la programación, otros muchos lenguajes de programación modernos, presentan una estructura de control llamada *switch*.

Sin embargo, dentro de Python no existe esta estructura de control concreta, sino que se resuelve con una estructura derivada de la anterior, llamada el *if*, que en Python se resume como *else if*.

La estructura de control y utilizando el *else if*, como podemos ver a continuación en el siguiente ejemplo, nos permite realizar una validación de múltiples casos verdaderos. Lo que realmente está haciendo, es preguntar, para cada uno de los días de la semana, si la información que ha introducido el usuario coincide, o no coincide, es decir, finalmente es una evaluación de verdadero o falso, con el dato de entrada.

En muchas ocasiones se ha discutido acerca de las razones por las cuales Python no contiene la misma estructura de control *switch* que contienen otros muchos lenguajes de programación contemporáneos, y en ocasiones se ha llegado a argumentar que la estructura de control *switch* no es más que un alias estético Para lo que, por dentro, un lenguaje de programación procesa, realmente como casos el id, con lo cual Python omite esta solución cosmética, para ofrecer al usuario la solución real.

```
dia = "lunes"

if dia == 'lunes':
    print("hoy es el peor dia de la semana")
elif dia == 'martes':
    print("hoy es el segundo peor dia de la semana")
elif dia == 'miercoles':
    print("ya estamos en el medio")
elif dia == 'jueves':
    print("ya casi es fin de semana")
elif dia == 'viernes':
    print("el mejor dia de la semana")
elif dia == 'sabado':
    print("hoy es fin de semana")
elif dia == 'domingo':
    print("y mañana es lunes")
else:
    print("yo no sé lo que has puesto, pero no es un dia de
```

```
la semana")
```

En este primer ejemplo de ejecución, podemos comprobar cómo hemos empezado creando una variable cuyo valor inicial es lunes, y a continuación empezamos a validar uno a uno los posibles días de la semana, especificando un caso falso que solo se ejecutará en el caso de que la variable introducida efectivamente no sea un día de la semana.

En este caso, comprobamos como el día que aparece en la pantalla es el lunes, y por lo tanto el caso a ejecutar corresponde al lunes.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============= RESTART:
/Users/josevicente/Desktop/Python/switch.py =============
hoy es el peor dia de la semana
```

A continuación, es interesante ilustrar el ejemplo anterior, pero en este caso introduciendo una palabra como cadena de caracteres dentro de la variable inicial, que no corresponde a ninguno de los días de la semana.

```
dia = "manzana"

if dia == 'lunes':
    print("hoy es el peor dia de la semana")
elif dia == 'martes':
    print("hoy es el segundo peor dia de la semana")
elif dia == 'miercoles':
    print("ya estamos en el medio")
elif dia == 'jueves':
    print("ya casi es fin de semana")
elif dia == 'viernes':
```

```
    print("el mejor dia de la semana")
elif dia == 'sabado':
    print("hoy es fin de semana")
elif dia == 'domingo':
    print("y mañana es lunes")
else:
    print("yo no sé lo que has puesto, pero no es un dia de
la semana")
```

Realizamos esta validación simplemente para comprobar que ninguno de los casos previstos dentro de la estructura de control se ejecuta como verdadero, y dado que ninguno de los casos es correcto, en ese caso el caso a ejecutar es el último, qué es el caso *else*.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
/Users/josevicente/Desktop/Python/switchdefault.py =========
yo no sé lo que has puesto, pero no es un dia de la semana
```

6.2.4. Try except

Una estructura de control muy especial es *try except*, ya que nos permite implementar métodos de evaluación de códigos relacionados con la programación defensiva.

La programación defensiva consiste en asumir que durante la ejecución de nuestro código pueden ocurrir problemas y esos problemas pueden generar errores, pero sin embargo, no queremos que nuestro código se detenga en esos errores.

Para esto, la estructura de *Try except* Intenta, en primer lugar, ejecutar un bloque de código, y en el caso de que no sea posible ejecutar ese bloque de código, o ese bloque de código de error, ejecuta, de forma eventual y opcional, aquello que hayamos introducido dentro del caso excepto.

Deberemos tener en cuenta que la ejecución de un programa en la época en la que vivimos depende de muchas circunstancias variables, que pueden no haber sido todas tenidas en cuenta, por una mera cuestión de imposibilidad, en el momento de planificar y programar la aplicación informática que estamos desarrollando.

Por ejemplo, en el caso de los programas que utilizan recursos tales como por ejemplo lectura y escritura de archivos, o conexión a base de datos, la utilización exitosa de esos recursos puede depender de muchos factores que pueden no tener que ver con el programa o con el código que hemos escrito.

Es por esto que la estructura de control condicional *try except* nos permite adaptarnos a esas circunstancias cambiantes, sin tener que sacrificar la estabilidad de la ejecución de nuestro código

```
print(x)
```

Al ejecutar este ejemplo de código, podemos comprobar que estamos solicitando que se imprima por pantalla una variable que previamente no ha sido definida, lo cual, evidentemente, genera un error en el intérprete.

La cuestión con respecto a este ejemplo consiste en que si hubiera más código a continuación de la línea dónde está el error, ese código nunca se llegaría a ejecutar, ya que el error para la ejecución del resto del programa.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

=============== RESTART:
```

```
/Users/josevicente/Desktop/Python/try.py ==============
Traceback (most recent call last):
  File "/Users/josevicente/Desktop/Python/try.py", line 1, in
<module>
    print(x)
NameError: name 'x' is not defined
```

Sin embargo, en el siguiente bloque de código que se presenta a continuación, la impresión de esa variable que previamente no ha sido definida, se ha introducido dentro de un bloque de excepción.

```
try:
    print(x)
except:
    print("ha ocurrido algún error")
```

Podemos comprobar, cuando ejecutamos el programa, que, aunque técnicamente debería haber un error, el error ha ejecutado el caso de excepción, y por lo tanto la ejecución del programa principal no se detiene, sino que puede continuar trabajando.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

=============== RESTART:
/Users/josevicente/Desktop/Python/try.py ==============
ha ocurrido algún error
```

6.3. Estructuras de bucle

6.3.1. Introducción

La propia esencia de los programas informáticos consiste en entender que existen para automatizar tareas y para hacer que la vida del ser humano sea menos tediosa. De esta forma, en muchas ocasiones en un programa informático nos encontramos escribiendo bloqueos de código de forma repetitiva.

Cuando esto ocurre, deberemos tener en cuenta que existen una serie de estructuras de control, llamadas estructuras de bucle, cuya finalidad consiste precisamente en automatizar todo ese código repetitivo, y en hacer Que sea el programa, el que trabaje, y no el usuario humano.

6.3.2. For

La estructura de control *For* nos permite repetir la ejecución de una porción del código una cantidad determinada de veces.

Dentro de la sintaxis de la invocación de esta estructura de control, encontramos una condición de inicio y una condición de finalización, en el ejemplo que mostramos a continuación, se ejecuta un código repitiéndolo 10 veces.

```
# for(int i = 1;i<10;i++){

for i in range(1,10):
    i = i + 1
    print("esta es la linea",i)

print("Si ves esto, es que ya no estamos en el bucle for")
```

Al ejecutar por primera vez este código, quizás puedas sorprenderte de ver aparecer, en la terminal o en la consola, 30 líneas de código.

Esto es lo que hace el bucle *for*, en el sentido de que este tipo de bucle nos permite realizar una tarea repetitiva, ejecutándola una única vez en el código fuente de nuestra aplicación.

Por lo tanto, ya que hemos indicado que queremos que el código se ejecute 10 veces, el sistema nos da, como resultado, 10 líneas en la terminal, cada una encadenando e incluyendo la variable de iteración que hemos puesto dentro de la declaración del bucle.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

=============== RESTART:
/Users/josevicente/Desktop/Python/for.py ==============
esta es la linea 2
esta es la linea 3
esta es la linea 4
esta es la linea 5
esta es la linea 6
esta es la linea 7
esta es la linea 8
esta es la linea 9
esta es la linea 10
Si ves esto, es que ya no estamos en el bucle for
```

6.3.3. While

Mientras que en la estructura anterior, en la propia declaración de la estructura, se introduce la condición de inicio y la condición de finalización, y de forma implícita la condición de incremento, que por defecto consiste en incrementar una unidad el iterador, la estructura de control *While*, el principio se ejecuta de forma continua e infinita, Aunque, sin embargo, podemos introducir la condición de inicio justo antes de declarar la

estructura, y debemos declarar la condición de incremento dentro de la propia estructura.

Deberemos tener en cuenta que al igual que en el resto de estructuras de control, es necesario que el código que va a ser ejecutado de forma repetitiva este sangrado con respecto a la declaración de la estructura, para que el sistema entienda que ese es el código que está contenido, y que por tanto será repetido.

El intérprete entenderá que la siguiente línea que ya no está sangrada, será la finalización del bucle.

```
numero = 1
while numero < 10:
    numero += 1
    print("hola")
```

Tras la ejecución del bloque de código Antonio, podremos comprobar como en la consola aparecen 10 repeticiones de la palabra hola.es importante que nos demos cuenta de que en el bloque de código anterior hemos dispuesto de la condición de inicialización fuera del bucle, de la condición de finalización dentro de la propia definición del bucle, y de la condición de incremento dentro del bucle.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============== RESTART:
/Users/josevicente/Desktop/Python/while.py =============
hola
hola
hola
hola
```

```
hola
hola
hola
hola
hola
```

7. Ejercicio práctico: calculadora

7.1. Presentación del ejercicio

A continuación, vamos a desarrollar un ejercicio, que consiste en la creación de una sencilla calculadora, cuyo objetivo es demostrar la aplicación práctica de, si no todos, al menos de una cantidad importante de los conocimientos que hemos adquirido hasta el momento.

El objetivo de este ejercicio es importante, y por eso merece un capítulo específico dentro de este libro, ya que es lo que se pretende no es solo poner en práctica esos conocimientos, sino demostrar cómo estructurar un programa, y que eso te pueda servir de referencia y de inspiración para futuros programas, y para aquellos programas que tú puedas hacer.

Lo primero que queremos es describir el funcionamiento del programa para entender cuál es el código que, a continuación, vamos a escribir.

Queremos crear una calculadora aritmética, que es un proyecto sencillo y sin demasiada ambición, pero que supone un paso adelante con respecto a la complejidad de los ejercicios que has desarrollado hasta el momento.

7.2. Estructura de un programa

Por tanto, el programa, a nivel de código, empezará con un comentario de presentación, a continuación, realizará una serie de preguntas al usuario, y en base a las respuestas que el usuario haya introducido, proporcionará un resultado.

La estructura de este programa, por lo tanto, es la misma que hemos comentado anteriormente al principio de esta publicación en el apartado destinado a comentar la estructura global de cualquier programa.

1. En primer lugar, crearemos una serie de comentarios para orientar a cualquier persona que en un futuro pueda abrir nuestro código
2. A continuación, requerimos al usuario que nos proporcione cierta información, que introduciremos en estructuras de control, tales como por ejemplo variables, y eso serán los datos de entrada del programa.

3. A continuación, el programa realizará una serie de cálculos matemáticos con los datos de entrada del usuario.
4. Y por último, el programa ofrecerá un resultado, una salida por pantalla, con respecto a los cálculos realizados en base a la entrada del usuario.

Por lo tanto, al realizar un ejercicio como este, ya no estamos pensando sólo en las instrucciones completas concretas del lenguaje de programación, tal y como hemos hecho hasta el momento, y desde luego, como haremos en próximos capítulos cuando volvamos a aprender piezas nuevas del lenguaje de programación.

Pero es bueno y es conveniente llegado este momento, parar un poco, y antes de seguir avanzando en nuestros conocimientos, demostrar como todo aquello que hemos aprendido hasta el momento son piezas que en un momento dado podemos unir para construir algo más grande Que un simple ejercicio, en la forma de un programa informático, ciertamente completo.

7.3. Comentario inicial

Y en el siguiente bloque se muestra un ejercicio práctico, en el cual se trata de aplicar todos aquellos conocimientos que hemos aprendido hasta el momento.

El ejercicio empieza creando un bloque de comentario, en el cual introducimos los datos esenciales del programa, tales como, por ejemplo, su nombre, el autor, el año de creación, y la versión que se presenta.

```
'''
Programa calculadora
(c) 2020 Jose Vicente Carratala
v0.1
'''
```

7.4. Solicitud de entrada: nombre del usuario

A partir de ese punto, el programa toma la entrada de ejecución por parte del usuario, preguntándole su nombre, tras lo cual utilizamos esa variable para dar la bienvenida al usuario, simplemente como primera demostración y punto de entrada al programa, para comprobar que podemos tomarle todo a ver usuario y guardarla de alguna forma.

```
'''
Programa calculadora
(c) 2020 Jose Vicente Carratala
v0.1
'''
## Bienvenida al programa
print("bienvenido al programa calculadora")
print("Introduce tu nombre")
nombre = input()
print("Hola",nombre,"te doy la bienvenida al programa
calculadora")
```

Al ejecutar el programa comprobamos cómo, en primer lugar, se le pide al usuario su nombre, para a continuación sacarlo por consola, de forma que podemos ver demostrado que tenemos la capacidad de solicitar información al usuario, y guardarla en forma de variables.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
/Users/josevicente/Desktop/Python/calculadora1.py ==========
bienvenido al programa calculadora
```

```
Introduce tu nombre
Jose Vicente
Hola Jose Vicente te doy la bienvenida al programa
calculadora
```

7.5. Opciones mostradas

Una vez que hemos pasado ese punto, a continuación le volvemos a preguntar de nuevo al usuario que operación desea realizar, pero para ello, antes, le presentamos las opciones en pantalla que va a poder elegir, donde la primera operación es una suma, la segunda operación es una resta, la tercera operaciones, una multiplicación, y la cuarta operación es la división.

```
'''
Programa calculadora
(c) 2020 Jose Vicente Carratala
v0.1
'''
## Bienvenida al programa
print("bienvenido al programa calculadora")
print("Introduce tu nombre")
nombre = input()
print("Hola",nombre,"te doy la bienvenida al programa
calculadora")
## Indica la operacion
print("Ahora elige la operación que vas a realizar"+
    "\n1.-Suma"+
    "\n2.-Resta"+
    "\n3.-Multiplicacion"+
    "\n4.-Division"+
      "")
```

Una vez presentadas estas opciones, realizamos tres preguntas al usuario, siendo la primera vez ellas la operación elegida, la segunda de ellas, el primero de los operandos, y la tercera de ellas la operación.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
/Users/josevicente/Desktop/Python/calculadora2.py ==========
bienvenido al programa calculadora
Introduce tu nombre
Jose Vicente
Hola Jose Vicente te doy la bienvenida al programa
calculadora
Ahora elige la operación que vas a realizar
1.-Suma
2.-Resta
3.-Multiplicacion
4.-Division
```

7.6. Entradas por parte del usuario

```
'''
Programa calculadora
(c) 2020 Jose Vicente Carratala
v0.1
'''
## Bienvenida al programa
print("bienvenido al programa calculadora")
print("Introduce tu nombre")
nombre = input()
print("Hola",nombre,"te doy la bienvenida al programa
calculadora")
```

```
## Indica la operación
print("Ahora elige la operación que vas a realizar"+
    "\n1.-Suma"+
    "\n2.-Resta"+
    "\n3.-Multiplicación"+
    "\n4.-Division"+
      "")
operacion = int(input())
print("La operación que has elegido es",operacion)
## Introduce dos números
print("Ahora introduce un número")
numero1 = int(input())
print("Ahora introduce otro numero")
numero2 = int(input())
```

Al ejecutar este programa podremos comprobar que, en primer lugar, podemos introducir el nombre dentro de la aplicación, tras lo cual el programa nos da la bienvenida utilizando nuestro nombre, es decir, la variable que acaba de tomar.

A continuación tenemos un mensaje de bienvenida, y se nos presenta el menú mediante el cual tenemos las opciones que podemos elegir.

Ahora, mediante dos instrucciones de entrada de información por parte del usuario, le preguntamos al usuario que introduzca un primer número, o lo que es lo mismo, un primer operador, y a continuación un segundo número, premio en cuenta que con estos dos operadores a continuación realizará las operaciones correspondientes.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
```

```
/Users/josevicente/Desktop/Python/calculadora3.py ==========
bienvenido al programa calculadora
Introduce tu nombre
Jose Vicente
Hola Jose Vicente te doy la bienvenida al programa
calculadora
Ahora elige la operación que vas a realizar
1.-Suma
2.-Resta
3.-Multiplicación
4.-Division
1
La operacion que has elegido es 1
Ahora introduce un numero
3
Ahora introduce otro numero
4
```

7.7. Selección de la operación a realizar

A partir de ese punto, en la siguiente parte del programa, en base a la operación elegida por el usuario, atrapamos la operación en un caso y con varios posibles, o bien, en un caso *If else*, y generamos un resultado, creando una operación aritmética de suma en el caso de que haya elegido la primera opción, una operación aritmética de resta, en el caso de que haya elegido la segunda opción, una operación aritmética de multiplicación, en el caso de que haya elegido la cuarta, tercera opción, y una operación aritmética de división, en el caso de que haya elegido la cuarta opción

```
'''
Programa calculadora
(c) 2020 Jose Vicente Carratala
v0.1
'''
## Bienvenida al programa
```

```
print("bienvenido al programa calculadora")
print("Introduce tu nombre")
nombre = input()
print("Hola",nombre,"te doy la bienvenida al programa
calculadora")
## Indica la operacion
print("Ahora elige la operación que vas a realizar"+
    "\n1.-Suma"+
    "\n2.-Resta"+
    "\n3.-Multiplicacion"+
    "\n4.-Division"+
      "")
operacion = int(input())
print("La operacion que has elegido es",operacion)
## Introduce dos numeros
print("Ahora introduce un numero")
numero1 = int(input())
print("Ahora introduce otro numero")
numero2 = int(input())

## Realiza la operacion
if operacion == 1:
    print("El resultado es:",(numero1+numero2))

if operacion == 2:
    print("El resultado es:",(numero1-numero2))

if operacion == 3:
    print("El resultado es:",(numero1*numero2))

if operacion == 4:
    print("El resultado es:",(numero1/numero2))
```

A continuación mostramos el resultado de la ejecución del programa anterior, realizando un volcado en la consola, para comprobar que el programa pedirá al usuario la información de inicio , la operación a realizar, los operadores, y por último devolverá el resultado en pantalla.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
/Users/josevicente/Desktop/Python/calculadora4.py ==========
bienvenido al programa calculadora
Introduce tu nombre
Jose Vicente
Hola Jose Vicente te doy la bienvenida al programa
calculadora
Ahora elige la operación que vas a realizar
1.-Suma
2.-Resta
3.-Multiplicacion
4.-Division
1
La operacion que has elegido es 1
Ahora introduce un numero
3
Ahora introduce otro numero
4
El resultado es: 7
```

8. Programación de funciones

8.1. Introducción a las funciones

En prácticamente cualquier sistema informático, la reutilización del código previamente escrito es un factor muy importante a la hora de desarrollar soluciones de software.

Disponemos de múltiples técnicas y estrategias para poder encapsular el código, como paso previo a su posterior reutilización.

Una de estas estrategias consiste en la utilización de funciones. Las funciones son burbujas de código, que contienen una entrada, una serie de operaciones, y una salida, y están pensadas específicamente para poder reutilizar se más adelante tantas veces como se quiera y se necesite.

8.2. Creación de funciones

En el siguiente ejemplo, mostramos cómo utilizar una función, definiéndose en primer lugar, y utilizándose en segundo lugar.

```
def miFuncion():
    print("Hola como estas")
    print("Hoy es miércoles")
    print("Hoy es verano")

miFuncion()
miFuncion()
miFuncion()
```

Al ejecutar este código podremos comprobar cómo, en un primer momento, definimos una función, para almacenarla en memoria y esperar a que el usuario ejecute dicha función.

A continuación llamamos a la función tres veces, y podemos comprobar cómo, cada una de las veces que vamos la función, se ejecutan las tres líneas de impresión contenidas dentro de la misma.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============ RESTART:
/Users/josevicente/Desktop/Python/funciones.py ===========
Hola como estas
Hoy es miércoles
Hoy es verano
Hola como estas
Hoy es miércoles
Hoy es verano
Hola como estas
Hoy es miércoles
Hoy es verano
```

8.3. Calculadora con funciones

Tomando como referencia el ejemplo que hemos desarrollado anteriormente, en el que hemos creado una calculadora, hasta el momento, la calculadora realizaba una ejecución, y una vez que realizaba la operación aritmética, la ejecución de nuestro programa informático finaliza.

Lo que hemos hecho es introducir el bloque principal del código dentro de una función llamada calculadora, y no debemos olvidar llamar a la función cuando finaliza el código.

Si nos fijamos, en el final del código, vemos que la llamada a la función se produce en dos momentos. En un primer momento se produce al final del todo, fuera de la función, como punto inicial de la ejecución del programa. Pero, sobre todo, lo más importante, es que podemos observar que también encontramos la llamada a la función como última línea de la propia función. Es decir, es una función recursiva, o lo que quiere decir que es una función que se llama así misma en el momento en el que termina.

De esta forma, entramos en un bucle infinito, en el que, una vez que el programa realiza su función, se vuelve a ejecutar asimismo para preguntarle al usuario que operación desea realizar a continuación.

Por supuesto una vez que nos encontramos en un bucle infinito en una consola de Python, podemos salir fácilmente de ese bucle infinito, simplemente pulsando la combinación de teclas, ctrl + C.

```
'''
Programa calculadora
(c) 2020 Jose Vicente Carratala
v0.1
'''
## Bienvenida al programa
print("bienvenido al programa calculadora")
print("Introduce tu nombre")
nombre = input()
print("Hola",nombre,"te doy la bienvenida al programa
calculadora")
def calculadora():
    ## Indica la operación
    print("Ahora elige la operación que vas a realizar"+
        "\n1.-Suma"+
        "\n2.-Resta"+
        "\n3.-Multiplicación"+
        "\n4.-Division"+
          "")
    operacion = int(input())
    print("La operación que has elegido es",operacion)
    ## Introduce dos números
    print("Ahora introduce un número")
    numero1 = int(input())
    print("Ahora introduce otro número")
    numero2 = int(input())

    ## Realiza la operación
    if operacion == 1:
```

```
        print("El resultado es:",(numero1+numero2))

    if operacion == 2:
        print("El resultado es:",(numero1-numero2))

    if operacion == 3:
        print("El resultado es:",(numero1*numero2))

    if operacion == 4:
        print("El resultado es:",(numero1/numero2))

    calculadora()

calculadora()
```

Al introducir las funciones dentro de nuestro programa, conseguimos encapsular todo el bloque principal de código dentro de una función a la que hemos dado el nombre de calculadora, y la llamamos de forma recursiva, haciendo que se vuelva a ejecutar cada vez que el bloque principal finaliza.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

====== RESTART:
/Users/josevicente/Desktop/Python/calculadorafunciones.py
======
bienvenido al programa calculadora
Introduce tu nombre
Jose Vicente
Hola Jose Vicente te doy la bienvenida al programa
calculadora
Ahora elige la operación que vas a realizar
```

```
1.-Suma
2.-Resta
3.-Multiplicación
4.-Division
1
La operación que has elegido es 1
Ahora introduce un número
3
Ahora introduce otro número
4
El resultado es: 7
Ahora elige la operación que vas a realizar
1.-Suma
2.-Resta
3.-Multiplicación
4.-Division
1
La operación que has elegido es 1
Ahora introduce un número
5
Ahora introduce otro número
6
El resultado es: 11
Ahora elige la operación que vas a realizar
1.-Suma
2.-Resta
3.-Multiplicación
4.-Division
```

8.4. Funciones con parámetros

Una función sin parámetros, como hemos visto hasta el momento, es un bloque de código que está condenado a ejecutarse siempre de la misma forma.

Sin embargo, en los paréntesis de la definición de una función, podemos introducir un número virtualmente ilimitado de parámetros, con el objetivo de que un mismo bloque de declaración de función no sirva para varios cometidos.

En el ejemplo que se presenta a continuación, tenemos una función que admite dos parámetros, que son el nombre y la edad. En función de esos dos parámetros, es cuando la función devuelve un mensaje personalizado, con lo cual, como vemos por último, podemos crear tantas variaciones de la función como sea necesaria, sin tener que declarar funciones diferentes.

```
def saludaMe(nombre,edad):
    print("Bienvenido,",nombre,"y tu edad es de",edad,"años")


saludaMe("Jose Vicente",30)
saludaMe("Juan",40)
```

Como podemos comprobar en la ejecución del programa las funciones con parámetros optimizan todavía mucho más su ejecución, ya que permiten que el contenido de la función sea personalizado con respecto a los parámetros que han sido introducidos. al ejecutar el código, por lo tanto, podemos ver que las dos ejecuciones son diferentes en la medida de que se adaptan a los parámetros introducidos al utilizar cada función

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

===== RESTART:
/Users/josevicente/Desktop/Python/funcionosconparametros.py
=====
Bienvenido, Jose Vicente y tu edad es de 30 años
Bienvenido, Juan y tu edad es de 40 años
```

9. Persistencia de la información

9.1. Introducción a la persistencia de la información

Hasta el momento, en todos los pequeños programas que hemos realizado, la información que contenía en el programa estaba almacenada en la memoria RAM del ordenador en el cual estamos ejecutando nuestros códigos.

Sin embargo, cada vez que nuestros programas finalizan su ejecución, el contenido de la memoria se borra, con lo cual no podemos guardar nada de lo que hemos realizado hasta ese momento de forma permanente.

Las operaciones relacionadas con la persistencia de la información, consisten en almacenar la información que utilicen nuestros programas en un soporte, tenga la naturaleza que tenga, que no sea volátil, y que por tanto pueda almacenar la información de forma consistente durante un periodo indeterminado de tiempo, especialmente cuando se retire la alimentación del sistema informático.

De los múltiples métodos de persistir la información, dentro de esta publicación, vamos a analizar tres de estos métodos. En primer lugar, el clásico método de escribir y leer archivos, y a continuación, la conexión con dos familias de bases de datos Estructuradas, como son Mysql y SQLite.

9.2. Lectura y escritura de archivos

Y la escritura y la lectura de archivos al disco duro, es decir, dentro del sistema de archivos del sistema operativo en el que estemos trabajando, es una de las formas más antiguas de persistir la información, y sin embargo, todavía hoy sigue siendo una de las más utilizadas.

A continuación presentamos las técnicas necesarias para poder escribir y leer archivos de texto desde Python.

9.2.1. Escribir archivo

Observaremos que el código necesario para poder escribir contenido en un archivo de texto, es muy sencillo, ya que en primer lugar utilizamos la función de Open para abrir un archivo, y a continuación establecemos un "*flag*", es decir, una bandera, para indicar en qué modo estamos abriendo el archivo, si lo abrimos en modo lectura, en modo escritura, en modo añadido, etc.

Una vez que hemos abierto el archivo, a continuación, utilizando la función *read*, podemos escribir tanto código como quedamos dentro de él.

Por último, es importante, siempre que abrimos un recurso externo al programa, cerrarlo correctamente, y no confiar con que el recurso se va a cerrar de forma automática.

Para esto es para lo que tenemos la instrucción y el método *Close*, para asegurarnos que cerramos manualmente el recurso que previamente hemos abierto.

```
archivo = open("miarchivo.txt",'a')
archivo.write("Este es un texto que estoy escribiendo")
archivo.close()
```

9.2.2. Leer archivo

De la misma forma que podemos escribir archivos, podemos igualmente leerlos.

La instrucción para abrir archivos se hace a través de la función *Open*, de la misma forma que hemos hecho cuando hemos querido escribir un archivo, con la diferencia de que en esta ocasión le introducimos una bandera específica para indicar que queremos leer el archivo.

Para leer cada una de las líneas del archivo, podemos utilizar directamente la función *readline*.

Sin embargo, para poder leer un número indeterminado de filas que puede contener el archivo que hemos abierto, podemos utilizar un buque *for*, que nos permitirá iterar una a una, las líneas contenidas dentro del archivo de texto, aunque no sepamos a priori cuántas puede llegar a ser.

```
archivo = open("miarchivo.txt",'r')

contador = 0
for i in range(0,10):
    contador += 1
    print(archivo.readline())
    if archivo.readline() == "" and contador > 5:
        break
```

Al ejecutar este código en el terminal, podremos comprobar que se nos devuelve el contenido del archivo de texto, finalizando la comprobación de que podemos realizar lecturas a archivos de texto plano.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============== RESTART:
/Users/josevicente/Desktop/Python/leer.py ==============
Este es un texto que estoy escribiendoEste es un texto que
estoy escribiendo
```

9.3. Fundamentos de SQL

SQL es un lenguaje para realizar búsquedas y operaciones en bases de datos relacionales. Las bases de datos relacionales han sido el estándar en cuanto a persistencia de la información en sistemas informáticos durante mucho tiempo, y es probable que lo sigan siendo durante mucho tiempo más. Siendo cierto que desde hace unos pocos años hay otra familia de bases de datos, NoSQL, que están ganando terreno, el sector sigue y seguirá necesitando almacenar información estructurada y relacionada, por lo que no existe, actualmente, una metodología que pueda desbancar, a corto plazo, a las bases de datos de tipo SQL.

Por este motivo, el aprendizaje del lenguaje SQL es fundamental por parte de cualquier programador, y además constituye una muy buena apuesta, en el sentido invertir tiempo aprendiendo un lenguaje que luego podrá amortizar sobradamente.

Por lo tanto, cualquier programador de aplicaciones debería tener una serie de conocimientos, al menos fundamentales, con respecto al trabajo en el lenguaje SQL y con respecto a la operación con bases de datos relacionales.

9.3.1. Introducción a SQL

El objetivo de este libro no es proporcionar una referencia exhaustiva en cuanto al uso y manejo de bases de datos relacionales, pero también se asume que el usuario puede no tener conocimientos previos en cuanto al manejo de estas bases de datos, y por lo tanto, el objetivo de este apartado, el libro consiste en proporcionar unos conocimientos mínimos y fundamentales, en cuanto al uso esencial del lenguaje SQL, para poder comprender en una mínima extensión, al menos cómo funcionan estas bases de datos, y para poder realizar los ejercicios que a continuación se proponen en este libro, en los cuales conectaremos el lenguaje de programación Python a dos bases de datos de tipo relacional.

9.3.2. Instalación de un entorno para desarrollar en SQL

9.3.2.1. Instalación de un servidor y acceso mediante la línea de comandos

Con respecto al motor de bases de datos MySQL, la forma más sencilla y más rápida de instalar un servidor de desarrollo dentro de nuestro ordenador local consiste en acudir a la siguiente dirección de Internet:

https://dev.mysql.com/downloads/

Desde la cual podremos descargar el servidor de MySQL e instalarlo en nuestro ordenador, independientemente del sistema operativo que utilicemos. En esta página podremos encontrar descargas para los principales sistemas operativos que están en uso hoy en día.

En el caso de que nos encontremos en sistemas operativos de tipo Linux, podemos utilizar comandos para la instalación del servidor MySQL desde los repositorios principales a los que tenga acceso esa distribución de Linux.

Por ejemplo, en un sistema operativo Linux. Sería de la siguiente forma:

```
sudo apt-get install mysql-server
```

Una vez que el servidor esté instalado, podremos acceder al mismo, abriendo una línea de comandos, o un terminal, introduciendo el siguiente comando, si el servidor está arrancado, y el acceso al mismo ha sido introducido en las rutas del sistema operativo.

```
mysql -u [usuario] -p
```

A partir de ese momento podremos comenzar a operar con la base de datos, introduciendo los comandos que se proponen en los siguientes ejercicios dentro de este apartado.

9.3.2.2. Instalación de una interfaz gráfica

Es fácilmente reconocible que trabajar con bases de datos en líneas de comando, puede ser una tarea ardua, especialmente cuando estamos comenzando nuestro camino como desarrolladores

Por lo tanto, existen diferentes herramientas que nos pueden ayudar a realizar una gestión inicial de las bases de datos, pero aportando una interfaz gráfica basada en ventanas, para poder hacer que nuestra operación con base de datos sea más fácil y más intuitiva.

En el caso de estar trabajando con MySQL, el programa recomendado para poder trabajar en modo escritorio es el oficial, MySQL Workbench, que podemos descargar desde la siguiente dirección.

https://www.mysql.com/products/workbench/

Existe otra herramienta con un nivel muy alto de oficialidad, que es phpMyAdmin, que nos habilita para poder gestionar bases de datos desde una interfaz web.

En el caso de querer utilizar la herramienta phpMyAdmin, es muy recomendable, instalar y utilizar un paquete llamado XAMPP, que podemos descargar e instalar de forma gratuita en cualquier sistema operativo de uso frecuente, y que, en un solo paquete, contiene tanto el servidor de bases de datos, como un servidor web local, como la herramienta phpMyAdmin para poder gestionar fácilmente las bases de datos que estemos desarrollando dentro de nuestro propio equipo.

Enlace de descarga de XAMPP:

https://www.apachefriends.org/es/index.html

9.3.3. Tipos de comandos o instrucciones en SQL

El propio lenguaje de búsquedas en base de datos, SQL contiene cinco categorías principales de instrucciones, que son las siguientes:

DDL - Data Definition Language

El lenguaje, definición de datos, nos permite crear entidades de bases de datos, nos permite crear bases de datos, tablas, nos permite eliminarlas, nos permite alterar su estructura, nos permite truncarlas, añadir comentarios, y renombrarlos

DQL - Data Query Language

En esta categoría cae la instrucción de selección, select, que nos permite realizar peticiones a una base de datos y recibir una serie de resultados como respuesta a nuestra consulta

DML - Data Manipulation Language

Dentro de esta categoría podemos encontrar los comandos más frecuentes para poder insertar nuevos registros dentro de una tabla, actualizar los registros existentes, y eliminar registros.

De forma adicional. También tenemos una instrucción para poder bloquear la tabla para poder controlar la concurrencia, o lo que es lo mismo, la cantidad de accesos simultáneos a esa pieza de información.

DCL - Data Control Language

En esta categoría encontramos fundamentalmente dos instrucciones, que nos permiten asignar permisos para que un usuario pueda tener acceso a un recurso, por ejemplo, una tabla, y también la instrucción que nos permite revocar dicho acceso

TCL - Transaction Control Language

Por último, dentro de esta categoría encontramos las instrucciones que nos permiten ejecutar una transacción, volver un paso atrás. En el caso de que la transacción haya devuelto un error, guardar puntos de control, y especificar las características de las transacciones.

A continuación, vamos a proporcionar una serie de ejemplos, a lo largo de un ejercicio guiado, del funcionamiento, de las principales instrucciones del trabajo con bases de datos, con el objetivo de poder proporcionar unos mínimos conocimientos, en cuanto al funcionamiento del lenguaje SQL, y

con el objetivo también de crear una base de datos Que sea de utilidad en los ejercicios que se van a presentar en las siguientes partes de este capítulo

9.3.4. DDL

9.3.4.1. Crear

9.3.4.1.1. Crear bases de datos

En primer lugar, y una vez que hayamos accedido al servidor de bases de datos, deberemos introducir el siguiente comando para crear una base de datos con el nombre curso Python.

Es muy importante tener en cuenta que las instrucciones en el lenguaje SQL deben ir separadas y finalizadas por;

```
CREATE DATABASE cursopython;
```

9.3.4.1.2. Usar una base de datos

Una vez que hayamos creado una base de datos, tanto si estamos usando las herramientas de línea de comando, como si estamos utilizando una herramienta de interfaz de usuario, debemos utilizar el siguiente comando para que el programa entienda que queremos utilizar la base de datos que acabamos de crear.

Esto es importante porque en muchas ocasiones damos por hecho Que solo por haber creado una base de datos automáticamente hemos entrado dentro de la misma y vamos a empezar a utilizarla, y en SQL esa asunción no es cierta.

```
USE cursopython;
```

9.3.4.1.3. Crear tablas en la base de datos

Una vez que hemos creado una nueva base de datos, y le hemos indicado al motor que queremos trabajar dentro de ella, a continuación tenemos que crear tablas para guardar la información dentro de la base de datos.

Para ello, utilizaremos el siguiente comando, que crea una tabla llamada personas, y a continuación crea cinco campos, que son los siguientes:

- Identificador
- Nombre
- Teléfono
- E-mail

Para cada uno de los campos especificamos el tipo de datos que van a contener, donde en este caso estamos usando dos tipos diferentes que son

- INT, para guardar números enteros
- VARCHAR para guardar cadenas de caracteres

En el caso de las cadenas de caracteres, entre paréntesis debemos indicar cuál es la longitud máxima de caracteres disponible para ese campo.

Para este ejemplo he introducido 100 caracteres de límite para cada uno de los campos, entendiendo que deberían ser suficientes para guardar cualquier nombre, cualquier teléfono, y cualquier correo electrónico

Por último, el campo identificador tiene unas características un poco especiales, y es que sirve como identificador inequívoco de cada uno de los registros, y por lo tanto tiene una serie de propiedades adicionales.

- NULL para especificar que ese campo no se puede dejar vacío
- Auto increment para especificar que el número del identificador debe crecer de forma automática, es decir, el sistema lo gestiona automáticamente y no hace falta que nosotros lo introduzcamos de forma manual
- Y por último, al final de la declaración se nombra al identificador como clave primaria, entendiendo que a partir de ese momento el identificador es el campo qué nombre de forma inequívoca a cada uno de los registros

```
CREATE TABLE personas (
    Identificador int NOT NULL AUTO_INCREMENT,
    nombre VARCHAR(100),
    telefono VARCHAR(100),
    email VARCHAR(100),
    PRIMARY KEY (Identificador)
)
```

9.3.4.1.4. Listar tablas

Una vez que hemos creado una tabla dentro de la base de datos, es interesante, especialmente cuando no estamos trabajando con una herramienta con interfaz de usuario, sino que podemos estar trabajando con la herramienta de línea de comandos, que podamos obtener un listado de las tablas contenidas en la base de datos, aunque sea simplemente para comprobar si la tabla que hemos creado en el paso anterior, se ha creado correctamente.

Para ello, podremos utilizar el siguiente comando Que, si todo va bien, nos dará un listado de tablas donde aparecerá la tabla que hemos creado en el paso anterior.

```
SHOW TABLES;
```

9.3.4.1.5. Listar tablas completo

Si queremos saber más acerca de la tabla que hemos creado anteriormente, podremos utilizar el siguiente comando, que nos dará información en pantalla acerca de los campos que contiene la tabla, el tipo de campo, la longitud, y otras características especiales que pueda contener, como, por ejemplo, en el caso del campo identificador, que es una clave primaria.

```
DESCRIBE personas;
```

9.3.4.2. Alterar tablas

Si nos hemos equivocado, al crear una tabla, podemos modificarla, o también, en muchas ocasiones, suele ocurrir que después de haber creado una tabla nos damos cuenta de que está incompleta y necesitamos modificar su estructura, por ejemplo para añadir más campos.

De esta forma, tenemos el comando alter, que nos permite, por ejemplo, en el siguiente Bloque de código, modificar la estructura de la tabla personas, para añadir un campo nuevo llamado dirección.

```
ALTER TABLE Customers
ADD Email varchar(255);
```

Hay veces en las que queremos alterar la estructura de una tabla para añadir Campus, y hay otras veces en las que necesitamos justo lo contrario, eliminar Campus o columnas de una tabla.

En este caso, podremos utilizar la siguiente sentencia SQL que nos permitirá eliminar las columnas que necesitemos de una tabla de la base de datos.

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Es importante anotar, llegado este punto, que mi recomendación personal consiste en realizar una copia de seguridad de la base de datos antes de realizar cualquier tipo de operación de eliminación.

Este consejo no se aplica tanto a esta base de datos que estamos creando en este momento, ya que, por el momento, es una base de datos vacía, y por tanto, las probabilidades de eliminación de datos de valor, son bastante escasas, sino que es una recomendación general orientada, sobre todo al momento en el que estemos trabajando con bases de datos reales en un entorno de producción.

Hay un refrán bastante pesimista pero a la vez bastante real, que dice que cualquier persona que esté aprendiendo en un momento u otro de su periodo de aprendizaje, acaba eliminando o borrando alguna base de datos de producción.

Dentro de que yo siempre he considerado qué ese refrán, desgraciadamente, es bastante realista, siempre recomiendo hacer copias de seguridad periódicas, y desde luego copias de seguridad excepcionales en el momento en el que se vaya a producir alguna alteración en la estructura de una base de datos.

9.3.4.3. Eliminar

De la misma forma que podemos eliminar un campo de una tabla de la base de datos, también podemos eliminar una tabla completa utilizando el siguiente comando.

```
DROP TABLE table_name;
```

En el mismo sentido que en el apartado anterior, he recomendado realizar copia de seguridad de una base de datos antes de realizar cualquier tipo de alteración de la estructura de la misma, especialmente cuando esa alteración consiste en eliminar una columna, el mismo consejo se puede aplicar para el caso de eliminar una tabla de la base de datos

9.3.4.4. Truncar

En las bases de datos de tipo relacional existen dos conceptos diferenciados que son la estructura y los datos de una tabla. La estructura hace referencia a la forma que tiene la tabla con respecto a los datos que va a contener, y los datos hace referencia al contenido real de una tabla.

En el caso anterior hemos visto cómo podemos eliminar una tabla, y cuando eso ocurre, obviamente se eliminan ambos tipos de información, se elimina la estructura de la tabla, y se eliminan también los datos contenidos en ella.

Sin embargo, hay muchas ocasiones en las que queremos vaciar los datos de una tabla, pero no queremos realmente eliminar la estructura, queremos que la estructura se quede en blanco, para poder rellenar nuevos datos.

Cuando eso ocurre, existe un comando llamado troncar, que nos permite realizar la operación de vaciado, eliminando los datos, pero sin alterar la estructura de la tabla

```
TRUNCATE TABLE table_name;
```

*Nota: En este libro se realizan ejemplos con MySQL y con SQLite. El comando TRUNCATE está disponible únicamente en MySQL - en SQLite se puede realizar un vaciado de tabla con una instrucción DELETE

9.3.5. DCL

Con esta parte del lenguaje SQL, podemos ser capaces de asignar y revocar permisos para que los usuarios puedan tener acceso, o podamos gestionar ese acceso, a las tablas o a las bases de datos contenidas en nuestro servidor.

En este caso, para estos ejemplos, y en este libro, vamos a trabajar únicamente con la instrucción de añadir permisos, que presentamos en el siguiente apartado.

9.3.5.1. Asignar permisos

Cuando queremos asignar permisos a los usuarios de la base de datos, podemos utilizar el siguiente comando para conseguir ese fin.

Con el comando que se presenta a continuación, se asignan todos los privilegios de acceso a una base de datos, a un usuario concreto.

Debemos tener en cuenta y recordar que el objetivo de esta publicación no es realizar una exposición extensiva de todas las opciones que tenemos en el lenguaje SQL, si no aprender lo mínimo Del trabajo con bases de datos relacionales, para poder combinar estas bases de datos con el lenguaje de programación Python que estamos aprendiendo. Si consultamos la

documentación oficial de mayo SQL, o bien consultamos, alguna publicación de naturaleza similar a este mismo libro, podremos comprobar, como, de hecho, existen multitud de opciones para poder asignar permisos de los usuarios a las bases de datos.

```
GRANT ALL PRIVILEGES ON database_name.* TO
'username'@'localhost';
```

9.3.6. DML

Esta categoría de instrucciones, de hecho, es la más conocida por la gran mayoría de usuarios, ya que contiene las instrucciones que se acaban utilizando en el día a día de la gestión real de una base de datos.

A continuación vamos a presentar las cuatro instrucciones principales para poder trabajar con cualquier base de datos, en lo que se conoce como un sistema CRUD, es decir, un sistema en el que se pueda

- Create, crear,
- Read, leer
- Update, actualizar
- Delete, eliminar

9.3.6.1. Seleccionar

Mediante el comando de selección realizamos una consulta a la base de datos en la que nos devuelve un conjunto de resultados.

En el siguiente ejemplo mostramos una instrucción SQL que nos devuelve todo el contenido de la tabla personas.

```
SELECT * FROM personas
```

Una vez más, recordamos que el objetivo de esta publicación no es realizar una exposición extensa de todas las capacidades del lenguaje SQL, sino

realizar una exposición mínima para que el usuario pueda trabajar los fundamentos de una base de datos relacional.

La selección de registros puede contener una gran cantidad de cláusulas adicionales, y de funciones programadas en el lenguaje SQL, que no son el objetivo de esta publicación, y que el usuario podrá encontrar en otras publicaciones de naturaleza similar a esta, pero orientados específicamente al aprendizaje del lenguaje SQL.

9.3.6.2. Insertar

A continuación, se presenta una consulta de inserción, cuyo objetivo consiste en introducir nuevos datos, es decir, nuevos registros dentro de una base de datos.

El siguiente ejemplo contempla la inserción de un registro dentro de la tabla personas que hemos creado anteriormente.

Es destacable comprobar dos cosas:

En primer lugar, Para el ejemplo propuesto, el número de campos que se insertan en la base de datos debe ser exactamente el mismo número de campos que hemos especificado anteriormente, que eran cinco.

Si hay una tabla que espera cinco columnas o campos, le introducimos cuatro datos, o le introducimos seis datos, la inserción fallará.

En la sintaxis que se presenta a continuación, por tanto, el número de CAMPOS o de columnas debe coincidir perfectamente.

La segunda de las consideraciones que deseo realizar es que como podemos comprobar el primero de los campos tiene un valor NULL en la inserción.

Esto es así porque anteriormente hemos definido que el primero de los campos, el identificador, es una clave primaria, numérica, y auto incremental. Al poner un valor nulo, le estamos permitiendo al motor de base de datos que gestione automáticamente, cuál es el identificador que tiene que asignar a ese registro.

```
INSERT INTO personas
VALUES (
NULL,
'Jose Vicente',
'12345',
'info@jocarsa.com'
)
```

9.3.6.3. Eliminar

A continuación, presentamos la instrucción de eliminación, que nos permite eliminar registros de la base de datos. Es muy importante notar que en este ejemplo hemos incluido el uso de la cláusula Ware, que nos permite añadir condiciones para poder acotar exactamente los registros que van a ser eliminados.

Si no existiera esta cláusula y ejecutáramos únicamente en la primera línea del siguiente ejemplo, lo que haría la petición a la base de datos sería eliminar todos los registros de la tabla personas.

Por lo tanto es importante incluir esa cláusula, para poder acotar los registros que vamos a eliminar, y no vaciar la tabla de la base de datos por accidente.

```
DELETE FROM personas
WHERE nombre = 'Jose Vicente'
```

9.3.6.4. Actualizar

Por último, la consulta de actualización nos permite modificar información de un registro previamente introducido.

En la siguiente consulta que se presenta. A continuación, encontramos tres líneas.

En la primera línea especificamos la tabla de la base de datos sobre la cual vamos a realizar la actualización.

En la segunda línea encontramos el campo que queremos modificar, y el valor que queremos introducir utilizando un operador de asignación.

En la tercera línea, de la misma forma que hemos hecho en el apartado anterior, en la eliminación de registros, utilizamos la cláusula *where* para acotar mejor la actualización y especificar sobre qué campos concretos queremos realizar esta operación de actualización de datos.

```
UPDATE personas
SET telefono = '1111'
WHERE nombre = 'Jose Vicente'
```

9.4. MySQL

MySQL es el motor de bases de datos más utilizado hoy en día.

Su conocimiento es muy importante independientemente del lenguaje de programación en el que nos encontremos. Saber cómo conectarnos a Mysql es importantísimo para conseguir persistir nuestra información en bases de datos de tipo SQL.

Este tipo de bases de datos relacionales nos permiten no solo almacenar nuestra información de una forma fiable, sino que pueden a nuestra disposición toda la potencia del lenguaje SQL para realizar cualquier tipo de consultas a los datos que previamente hayan sido almacenados.

9.4.1. Instalación del conector

En primer lugar, antes de empezar a escribir programas que se conectan a una base de datos de tipo MySQL, es necesario cumplir con dos requisitos.

En primer lugar, en nuestro ordenador debe existir, instalada, una copia del motor de bases de datos MySQL, y, en el momento de escribir y ejecutar nuestros scripts, el motor debe estar arrancado y en funcionamiento.

a continuación, es importante instalar el conector de mayo SQL para Python, y para poder realizarlo, deberemos arrancar un terminal, y dentro del terminal introduciremos el siguiente comando

Una vez que hayamos realizado esa operación, es cuando podremos escribir los programas que sean necesarios para conectarnos al servidor de MySQL, desde el lenguaje de programación Python

El conector puede ser descargado desde:

https://dev.mysql.com/downloads/connector/python/

9.4.2. Importar el conector

En primer lugar, vamos a conectarnos a una base de datos, y para ello veremos que el siguiente programa presenta la siguiente estructura

```
import mysql.connector as my

mibd = my.connect(
    host = "localhost",
    user = "josevicente",
    password = "josevicente"
    )

print(mibd)
```

En la primera línea invocamos a la librería que previamente hemos instalado en cuanto a conexiones con bases de datos de tipo MySQL, para, a continuación, utilizar el método *Connect*, y almacenar la conexión en una variable.

Almacenar la conexión en una variable es muy importante para poder utilizar dicha variable más adelante, y poder lanzar peticiones dentro de esa estructura de datos.

Podremos comprobar que Connect es un método que requiere una serie de parámetros, tales como por ejemplo el host, que es el servidor, al cual nos conectamos, el usuario, y la contraseña.

En este caso, como en la gran mayoría de casos, en los cuales realizamos un desarrollo en nuestro propio ordenador, el huésped que aloja la base de datos, será denominado como local host.

Más adelante, cuando desarrollemos aplicaciones que estén alojadas en servidores remotos, tendremos que consultar con el administrador del servidor para que nos facilite estos datos de conexión.

Es muy importante resaltar en este punto del ejercicio que para poder conectarnos correctamente a una base de datos MySQL no solo es importante que dicha base de datos esté creada, sino que también es importante que haya sido creado un usuario, con una contraseña, y que tenga permisos para poder acceder a esa base de datos.

9.4.3. Comprobar la conexión

El puerto más común en el que se instala y se ejecuta el servidor de MySQL es el 3306.

Sin embargo, en ocasiones puede ocurrir que, por diferentes tipos de necesidades, el servidor donde se ejecuta MySQL necesita que el puerto 3306 quede libre, y por lo tanto puede ser asignado a cualquier otro puerto diferente.

En el ejemplo anterior, el código que habíamos desarrollado no indicaba el puerto de conexión, y por lo tanto, cuando esto ocurre, se asume que el puerto de conexión es el puerto por defecto, el 3306.

Sin embargo, en el caso de que por los motivos que sean, el puerto haya cambiado, podemos indicarlo en el momento de realizar la conexión, introduciendo un nuevo parámetro llamado Port en la función Connect que habíamos utilizado anteriormente.

```
import mysql.connector as my
```

```
mibd = my.connect(
    host = "localhost",
    port = "3306",
    user = "josevicente",
    password = "josevicente"
    )

print(mibd)
```

Al ejecutar este programa, al menos hasta el punto en el que lo hemos desarrollado, podremos comprobar como, si la conexión es exitosa, y se cumplen las siguientes condiciones 2 puntos

- Tenemos un servidor de bases de datos MySQL en marcha
- el servidor está escuchando en el puerto 3306
- existe un usuario creado dentro del servidor llamado José vicente
- dicho usuario tiene una contraseña llamada igualmente José Vicente

Si estas condiciones se cumplen, podremos comprobar como el servidor lanzará un mensaje de conexión aceptada, lo cual quiere decir que podremos continuar con las siguientes partes del ejercicio.

Las conexiones a base de datos de tipo MySQL son complejas siempre las primeras veces que las realizamos, ya que cualquier detalle que nos hayamos dejado, será suficiente como para no tener una respuesta positiva por parte del servidor.

Cuando esto ocurre simplemente lo que debemos hacer es perseverar en nuestros intentos por conectar, revisando y comprobando que todos los datos estén correctos, hasta que el servidor no nos rechaza la conexión.

9.4.4. Realización de una inserción

Hasta el momento, nos habíamos conectado a el motor de bases de datos L, pero no habíamos especificado cuál es la base de datos a la cual queríamos acceder. Es por esto que podemos especificar la base de datos sobre la cual

vamos a trabajar, introduciendo el parámetro correspondiente dentro de la misma función Connect que hemos utilizado hasta el momento.

Una vez que nos hemos conectado, es cuando vamos a empezar a lanzar peticiones a la base de datos.

En primer lugar es importante tener en cuenta qué la librería nos va a invitar a utilizar la estructura de control. Trae excepto para realizar conexiones a base de datos.

Esta práctica no es nada infrecuente, se da en otros lenguajes de programación muy diferentes a Python, y tiene como objetivo prevenir cualquier error que pueda ocurrir durante la conexión con la base de datos.

Deberemos tener en cuenta que los errores de conexión a bases de datos son muy importantes y muy frecuentes, ya que en definitiva son dos piezas de software que corren de forma completamente independiente, y en muchas ocasiones, se ejecutan en máquinas diferentes, con lo cual los errores de conexión son, como decía anteriormente, más frecuentes de lo que podríamos esperar.

A continuación, una vez que hemos especificado esa estructura de control de captura de excepciones, creamos un cursor, y a continuación ejecutamos una petición a la base de datos, introduciendo la sentencia SQL dentro de un comando *execute*.

Por último, utilizamos el comando comité para lanzar la petición a la base de datos, y finalizar la ejecución de la misma.

Como última parte de este programa, en la función de excepción, lanzamos un print para el caso de que haya habido algún problema con la conexión a la base de datos.

```
import mysql.connector as my
try:
    mibd = my.connect(
        host = "localhost",
        port = "3306",
        user = "josevicente",
```

```
        password = "josevicente",
        database = "cursopython"
        )

    ##print(mibd)

    micursor = mibd.cursor()

    micursor.execute("INSERT INTO personas VALUES
(NULL,'jose','12345','jose@correo.com')")
    mibd.commit()

except:
    print("ha ocurrido algun error en la base de datos")
```

Al ejecutar este código, podremos comprobar , si la conexión a la base de datos se ha realizado de forma exitosa, qué el sistema habrá introducido un registro dentro de la tabla personas de la base de datos que estamos utilizando dentro de este curso.

deberemos tener en cuenta que cuando realizamos una inserción en la base de datos, no necesariamente tiene porqué salir algo en la consola, ya que de todas las funciones, de las cuatro principales, con las cuales operamos en bases de datos, la única que devuelve resultados por consola es la de selección.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

===== RESTART:
/Users/josevicente/Desktop/cursodepython/python065-mysql4.py
====
```

9.4.5. Selección de registros

Una vez que hemos realizado una conexión exitosa a la base de datos, y de hecho, en el ejercicio anterior, hemos conseguido insertar un registro dentro de la tabla de usuarios, vamos a continuación a realizar una petición de selección.

Para ello, el código de la aplicación es prácticamente igual que en el ejercicio anterior, salvo que en la ejecución, lanzamos una sencilla consulta SQL para obtener todos los registros de la tabla de personas.

La novedad en este caso consiste en que, una vez que hemos lanzado la ejecución, creamos una nueva variable, o una nueva estructura de datos, y dentro de ella almacenamos el resultado de llamar a la función Fake All.

La función obtiene todos los resultados derivados de la petición que hemos realizado, y los introduce en una estructura de datos complejos, para que más adelante la podamos recorrer.

A continuación, utilizando un sencillo bucle for, podemos recorrer la estructura de datos que acabamos de utilizar, para devolver en la pantalla todos y cada uno de los registros que provienen de la base de datos.

Debemos tener en cuenta que, de las cuatro operaciones básicas que podemos utilizar dentro de una base de datos, que son la inserción, la selección, la actualización, y la eliminación, solo la selección requiere procesar los datos que devuelve el servidor de bases de datos.

Las otras tres consultas no devuelven ninguna información, y por lo tanto no es necesario utilizar la estructura que presentamos en esta parte del ejercicio.

```
import mysql.connector as my
try:
```

```
    mibd = my.connect(
        host = "localhost",
        port = "3306",
        user = "josevicente",
        password = "josevicente",
        database = "cursopython"
        )

    ##print(mibd)
    micursor = mibd.cursor()
    micursor.execute("SELECT * FROM personas")
    miresultado = micursor.fetchall()
    ##print(miresultado)
    for i in miresultado:
        print("tengo un resultado que es:")
        print(i[1])
except:
    print("ha ocurrido algún error en la base de datos")
```

Al ejecutar el código anterior, podremos comprobar que , siempre que existan registros dentro de la tabla personas de la base de datos, estos registros saldrán en la pantalla. en un primer momento los estamos extrayendo a través de la consola, pero más adelante, una vez que aprendemos a trabajar con interfaces de usuario, podremos extraerlo de formas más gráficas

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

===== RESTART:
/Users/josevicente/Desktop/cursodepython/python064-mysql3.py
====

===== RESTART:
```

```
/Users/josevicente/Desktop/cursodepython/python064-mysql3.py
====
tengo un resultado que es:
jose
```

9.4.6. Operación de actualización

A continuación, dentro de esta parte del ejercicio, presentamos la posibilidad de realizar una consulta de actualización, realizando una variación del primer ejercicio de esta serie, en el cual hemos realizado una inserción, pero únicamente variando la sentencia SQL para actualizarla a una actualización de tipo SQL.

```
import mysql.connector as my
try:
    mibd = my.connect(
        host = "localhost",
        port = "3306",
        user = "josevicente",
        password = "josevicente",
        database = "cursopython"
        )

    ##print(mibd)

    micursor = mibd.cursor()

    micursor.execute("UPDATE personas SET telefono =
'987654'")
    mibd.commit()

except:
    print("ha ocurrido algun error en la base de datos")
```

Al ejecutar la operación de actualización en la terminal, podemos comprobar cómo, al igual que la operación de inserción, la operación de actualización no

tendrá ningún resultado concreto en la consola, a menos que la conexión falle, y obtengamos el resultado de ejecutar la cláusula *except*.

Deberemos entrar al gestor de base de datos para comprobar si la actualización se ha realizado de forma correcta.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

===== RESTART:
/Users/josevicente/Desktop/cursodepython/python066-mysql5.py
====
```

9.5. SQLite

Junto con MySQL, el motor de bases de datos. SQLite es uno de los motores más usados y más frecuentes cuando queremos combinar un programa informático con un sistema de almacenamiento de datos.

El propio nombre de esta base de datos puede resultar bastante confuso, ya que parece sugerir que únicamente sirve para almacenar pequeñas bases de datos, pero la realidad, especialmente si consultamos la documentación oficial de este motor de bases de datos, es que tiene una gran potencia con respecto a la sencillez, mediante la cual nos podemos conectar a este motor.

Debemos tener en cuenta que presenta una serie de diferencias con respecto al trabajo con motores de bases de datos como MySQL.

En primer lugar, es un motor de bases de datos basado en un único archivo. Esto quiere decir que, a diferencia de la base de datos que hemos trabajado anteriormente, utilizando este motor, como vamos a poder comprobar en los ejercicios siguientes, se crea un archivo que contiene la base de datos con la

cual vamos a poder trabajar. En un momento dado esto puede hacer que el trabajo con ese tipo de bases de datos y la migración de aplicaciones de un ordenador a otro sea bastante más rápido y más fácil que utilizando otros motores de bases de datos.

Sin embargo, también hay que tener en cuenta que estaba hace datos, no presenta, de forma nativa, la posibilidad de conectarse utilizando un usuario y una contraseña, lo cual hace que su uso se llama sencillo para usuarios noveles, pero también, intrínsecamente, esta circunstancia hace Que sea una base de datos con menos nivel de seguridad que, por ejemplo, vaya SQL.

9.5.1. Comprobación de la conexión

Lo que tenemos que hacer en primer lugar es comprobar si nuestra distribución de Python tiene previamente instalado el módulo para conectarnos a SQLite.

En el caso de que nuestra distribución no tenga instalado ese conector, podemos instalarlo fácilmente invocando a un terminal o a una consola, introduciendo el siguiente comando.

Una vez que nos hayamos asegurado de que nuestro sistema tiene soporte para trabajar con SQlite, Podremos introducir el código que indico a continuación, para comprobar el soporte que tenemos para la versión que tengamos instalada del motor de bases de datos.

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys
##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")
##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()
##Le pido algo a la base de datos en lenguaje SQL
cursor.execute("SELECT SQLITE_VERSION()")
##Datos contiene lo que me devuelve la base de datos
```

```
datos = cursor.fetchone()
print("La versión de SQLite es:",datos)
```

Si ejecutamos el código y el conector de Python está correctamente instalado en nuestro equipo, veremos que en lugar de lanzar un error, tenemos la versión del conector de ese culito para Python correctamente mostrado en la terminal.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========= RESTART:
/Users/josevicente/Desktop/Python/sqlitecomprobar.py
========
La versión de SQLite es: ('3.38.4',)
```

9.5.2. Inserción de registros

La primera de las operaciones de bases de datos con SQLite con las que vamos a trabajar consiste en una inserción.

En primer lugar, dentro del programa de ejemplo que vamos a desarrollar, importamos la librería de conexión a bases de datos para poder utilizar las funciones y todo el código incluido dentro de esta librería.

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys
```

Una vez que hemos comprobado que importamos exitosamente la librería, realizamos una conexión, abriendo el archivo de bases de datos llamado agenda. SQLite

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys

##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")
```

Una vez que hemos creado la conexión, establecemos un cursor para que el puntero se ponga en el punto correcto de la base de datos para empezar a operar.

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys

##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")

##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()
```

Y sobre ese cursor ejecutamos una petición SQL, que en este caso consiste en una inserción dentro de la tabla de contactos de una serie de cuatro valores.

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys
```

```
##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")

##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()

##Le pido algo a la base de datos en lenguaje SQL
cursor.execute("INSERT INTO contactos
VALUES(NULL,'Jorge','222222','jorge@correo.com');")
```

Por último, para ejecutar el código que hemos escrito, utilizamos el comando commit

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys

##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")

##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()

##Le pido algo a la base de datos en lenguaje SQL
cursor.execute("INSERT INTO contactos
VALUES(NULL,'Jorge','222222','jorge@correo.com');")

conexion.commit()
```

La ejecución del código anterior no devolverá necesariamente un resultado por pantalla, ya que hemos realizado una operación de inserción dentro de la base de datos, y por lo tanto la mejor forma de comprobar que la inserción

se ha realizado de forma correcta, será abrir la propia base de datos con una herramienta de gestión visual.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

== RESTART: /Users/josevicente/Downloads/Codigo
2/python035-SQLiteinsertar.py ==
```

9.5.3. Selección de registros

Una vez que hemos realizado la operación de inserción, la siguiente operación que vamos a realizar es la de lectura, es decir, selección de registros dentro de la tabla.

Para ello en primer lugar empezamos igual que hemos empezado con el ejercicio anterior, importando las librerías correspondientes, realizando una conexión, y creando un cursor dentro de la conexión.

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys

##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")

##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()
```

El siguiente paso consiste en la creación de una consulta SQL, mediante la cual solicitamos una selección de todo aquello que tenga la tabla de contactos.

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys

##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")

##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()

##Le pido algo a la base de datos en lenguaje SQL
cursor.execute("SELECT * FROM contactos;")
```

A diferencia de la operación de selección con respecto a las otras tres operaciones principales que son la inserción, la actualización, y la eliminación, es que la operación de selección es la única de las cuatro que devuelve un resultado, y por lo tanto debemos ser capaces de procesar ese resultado.

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys

##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")

##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()
```

```
##Le pido algo a la base de datos en lenguaje SQL
cursor.execute("SELECT * FROM contactos;")

##Datos contiene lo que me devuelve la base de datos
datos = cursor.fetchall()
```

Por eso a continuación, creamos una variable llamada datos, en la cual vamos a alojar una lista, conteniendo todos los elementos de cada una de las filas que devuelve, como resultado, la petición que hemos realizado a la base de datos.

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys

##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")

##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()

##Le pido algo a la base de datos en lenguaje SQL
cursor.execute("SELECT * FROM contactos;")

##Datos contiene lo que me devuelve la base de datos
datos = cursor.fetchall()

for i in datos:
    print("ID:",i[0],"\t nombre:",i[1],"\t telefono:
",i[2],"\t email:",i[3])
```

Por último, una vez que tenemos la información contenida en esa estructura de datos, utilizaremos una estructura de control *for* para recorrer uno a uno los registros que ha devuelto la conexión a la base de datos, y mostrarlos por pantalla utilizando una sencilla instrucción print

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys

##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")

##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()

##Le pido algo a la base de datos en lenguaje SQL
cursor.execute("SELECT * FROM contactos;")

##Datos contiene lo que me devuelve la base de datos
datos = cursor.fetchall()

for i in datos:
    print("ID:",i[0],"\t nombre:",i[1],"\t telefono:
",i[2],"\t email:",i[3])
```

Al ejecutar este código podremos comprobar como, en la consola, obtenemos un listado de los registros que contiene la base de datos, obteniendo tanto el identificador, como el nombre, como el teléfono, como el email.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.
```

```
==== RESTART: /Users/josevicente/Downloads/Codigo
2/python034-SQLiteleer.py ====
ID: 1   nombre: Juan        telefono:  123456   email:
info@juan.com
ID: 3   nombre: Jorge       telefono:  222222   email:
jorge@correo.com
```

9.5.4. Actualización de registros

La ejecución de una consulta de actualización es realmente sencilla desde el punto de vista en el que simplemente tenemos que reutilizar el mismo código que creamos anteriormente para poder realizar una operación de inserción, cambiando la consulta SQL, que pasamos como cadena durante la ejecución dentro del cursor.

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys

##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")

##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()

##Le pido algo a la base de datos en lenguaje SQL
cursor.execute("UPDATE contactos SET telefono = '123456';")

conexion.commit()
```

Al igual que ocurre en las operaciones de inserción y eliminación, realizar una actualización en la base de datos no genera necesariamente ningún resultado en la consola, por lo cual el resultado en consola estará vacío, y

deberemos abrir la base de datos en alguna aplicación de gestión visual, para comprobar que los cambios se hayan realizado correctamente

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

= RESTART:
/Users/josevicente/Desktop/cursodepython/python036-SQLiteact
ualizar.py
```

9.5.5. Eliminación de registros

Lo mismo ocurre con la consulta de eliminación, ya que, al igual que en el caso de la actualización y la inserción, no devuelve ningún resultado que procesar línea línea, y por tanto simplemente podemos utilizar el código desarrollado en los ejercicios de inserción y de actualización, Modificando la consulta para que refleje una eliminación de registro en el lenguaje SQL

```
##Para poder trabajar con bases de datos
import sqlite3 as lite
import sys

##Me conecto a una base de datos llamada agenda
conexion = lite.connect("agenda.sqlite")

##Establezco un cursor para saber en que punto de la base de
datos voy a trabajar
cursor = conexion.cursor()

##Le pido algo a la base de datos en lenguaje SQL
cursor.execute("DELETE FROM contactos WHERE Identificador =
2;")

conexion.commit()
```

Igualmente como en los casos anteriores, una operación de eliminación de registros no devuelve ningún resultado en la consola. deberemos acceder algún programa de gestión visual de bases de datos, para poder comprobar que los cambios se hayan realizado correctamente

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

= RESTART:
/Users/josevicente/Desktop/cursodepython/python036-SQLiteact
ualizar.py
```

10. Ejercicio práctico

10.1. Programa principal

A continuación, vamos a desarrollar un ejercicio práctico, en el cual estaremos aplicando todos los conocimientos que hemos aprendido hasta el momento, o al menos la mayor parte de ellos que podamos, para crear un programa que pueda tener una utilidad, que pueda tener un sentido, y que pueda ser representativo de los programas que en cualquier momento dado podemos estar interesados en desarrollar.

Para ello hemos elegido un programa cuya misión es guardar datos, y cuya misión es ofrecer al usuario poder realizar diferentes operaciones con esos datos.

Por supuesto, uno de los objetivos de este programa informático consiste en ofrecer al usuario la posibilidad de persistir la información de tal forma que sea recuperable más adelante, convirtiéndolo en un programa realmente de utilidad.

Para ello, para este programa y para este ejemplo, hemos elegido el método de persistencia en archivos de texto, ya que es el método más sencillo para realizar la operación de persistencia de los datos.

Para este ejercicio, por tanto, crearemos una agenda de contactos que contendrá tres piezas de datos, que serán el nombre, el teléfono, y el e-mail.

Esto es importante desde el punto de vista de que antes de empezar un programa informático, es recomendable que definamos algo llamado " el modelo de datos ". el modelo de datos consiste en definir la información que contendrá el programa, y en definitiva, la información que viajará a todas partes, es decir, la información que se le mostrará al usuario, la información con la que trabajará el programa por dentro, y la información que se persistirá Dentro de la base de datos o cualquiera que sea el mecanismo de persistencia que utilicemos.

Aunque el programa informático que vamos a desarrollar a continuación es realmente sencillo, en metodologías de desarrollo, aquella parte del programa que muestra datos al usuario, recibe el nombre de vista, aquella parte del programa que realiza un procesamiento interno, recibe el nombre de controlador, y aquella parte Del programa, que se encarga de la

persistencia de datos, acaba recibiendo, aunque muchas veces sea de forma errónea, el nombre de modelo, haciendo referencia al modelo de datos.

Por lo tanto, vamos a empezar el programa de la forma más sencilla, que es introduciendo un comentario para darle título a la aplicación que vamos a desarrollar.

```
##Programa agenda por Jose Vicente Carratala
```

A continuación, creamos una estructura de datos consistente en una lista de dos dimensiones, que contiene algo que podríamos llamar la cabecera del modelo de datos, es decir, las columnas que indican la información que se va a guardar dentro de esa estructura.

Para ello podemos comprobar como al definir la estructura de datos de agenda, estamos introduciendo dos corchetes tanto para abrir como para cerrar, indicando que lo que estamos creando no es una lista de una dimensión, sino que es una lista de dos dimensiones.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]
```

En la siguiente parte del programa vamos a crear un menú mediante el cual solicitaremos al usuario los elementos que van a formar parte de un nuevo registro dentro de la agenda, es decir, le pediremos al usuario el nombre a introducir, y lo pondremos dentro de una variable, utilizando una entrada de tipo input, y realizaremos el mismo proceso para solicitarle tanto el teléfono como el correo.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]
```

```
def miAgenda():
    print("Introduce el nuevo nombre en la agenda")
    nombre = input()
    print("Introduce el numero de telefono")
    telefono = input()
    print("Introduce el correo")
    correo = input()
```

A continuación evidentemente introduciremos dicha información dentro de la estructura de datos de agenda, y simplemente por una cuestión de comprobación imprimimos todo el contenido de la agenda hasta ese momento.

Una de las características principales de esta parte del ejercicio es que introducimos todo este código dentro de una función, con lo cual, al finalizar, la función, llamamos a la propia función.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]


def miAgenda():
    print("Introduce el nuevo nombre en la agenda")
    nombre = input()
    print("Introduce el numero de telefono")
    telefono = input()
    print("Introduce el correo")
    correo = input()
    # antes de hacer nada mas, lo metemos en la lista, y
sacamos la lista
    agenda.append([nombre,telefono,correo])
    print(agenda)
    # ejecucion recursiva
```

```
    miAgenda()
```

Esto quiere decir que cuando la función se ejecuta, al finalizar su ejecución, se llama así misma, lo que quiere decir que el programa entra en un bucle infinito, pero en un bucle infinito controlado, es decir, no bloquea la ejecución, sino que simplemente, al finalizar la inserción de cada registro, el programa vuelve a comenzar desde cero.

Deberemos tener en cuenta que si deseamos detener la ejecución del programa podemos hacerlo sin ningún tipo de problema pulsando la combinación de teclas control+C dentro de la terminal donde se está ejecutando el programa.

Si ejecutamos el programa hasta este punto veremos que realmente no ocurre nada, porque hemos definido una función, pero nunca la hemos llamado hasta el momento. Por lo tanto, el programa tiene que finalizar llamando a la función para iniciar la ejecución de dicha función, y que una vez que la ejecución ha comenzado, el programa, es decir, la función, se puede llamar así misma en bucle una vez que finaliza cada nueva inserción dentro de la estructura de datos.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]


def miAgenda():
    print("Introduce el nuevo nombre en la agenda")
    nombre = input()
    print("Introduce el numero de telefono")
    telefono = input()
    print("Introduce el correo")
    correo = input()
    # antes de hacer nada mas, lo metemos en la lista, y
sacamos la lista
```

```
    agenda.append([nombre,telefono,correo])
    print(agenda)
    # ejecucion recursiva
    miAgenda()

miAgenda()
```

Al ejecutar este programa podremos comprobar cómo el sistema ejecuta la función principal de forma recursiva, y podremos introducir tantos registros como sea necesario dentro de la estructura de datos de la agenda. únicamente deberemos tener en cuenta que, al menos hasta este momento, no tenemos todavía una forma de guardar los datos de forma permanente, con lo cual toda aquella información que hayamos guardado se perderá en el momento en el que salgamos del programa.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============= RESTART:
/Users/josevicente/Desktop/Python/agenda.py =============
Introduce el nuevo nombre en la agenda
Jose Vicente
Introduce el numero de telefono
12345
Introduce el correo
info@jocarsa.com
[['nombre', 'telefono', 'email'], ['Jose Vicente', '12345',
'info@jocarsa.com']]
Introduce el nuevo nombre en la agenda
Juan
Introduce el numero de telefono
12345
Introduce el correo
juan@jocarsa.com
```

```
[['nombre', 'telefono', 'email'], ['Jose Vicente', '12345',
'info@jocarsa.com'], ['Juan', '12345', 'juan@jocarsa.com']]
Introduce el nuevo nombre en la agenda
```

10.2. Persistencia: escritura

La versión anterior del programa que hemos desarrollado, es ciertamente, imperfecta desde el punto de vista en el que los datos únicamente existen en la memoria RAM del ordenador, mientras que el programa está en funcionamiento, pero una vez que finalizamos la ejecución del programa, la memoria se vacía y perdemos todos los datos que hayamos introducido anteriormente.

Como evidentemente, esto es algo bastante incómodo, en la siguiente parte del ejercicio, vamos a introducir un método de persistencia, mediante el cual, partiendo del código que hemos desarrollado en la parte anterior del ejercicio, introducimos una serie de líneas después de la impresión de la agenda completa, mediante las cuales abrimos un archivo llamado agenda.txt, en modo de añadido, forma te amos un texto que incluya tanto el nombre como el teléfono como el correo, y lo escribimos dentro de ese archivo, tras lo cual lo cerramos.

Hay que tener en cuenta que, si nos fijamos en el formato que presenta la cadena que persistimos dentro del archivo de texto, veremos que estamos separando tanto el nombre, como el teléfono, como el correo, utilizando caracteres de coma, e introduciendo \N al finalizar la línea, para forzar que el documento de texto salto de línea después de haber introducido cada uno de los registros.

al realizar esta operación, cada vez que guardamos un nuevo registro, es decir, cada vez que le preguntamos al usuario final por la introducción de un nuevo registro, este registro queda persistido en un archivo de texto, aunque, cuando cerramos el programa y lo volvamos a abrir, no podemos ver todavía todos aquellos registros que han sido guardados en el archivo de texto, aunque sí que podríamos verlos si abrimos directamente el archivo de texto propiamente dicho.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]


def miAgenda():
    print("Introduce el nuevo nombre en la agenda")
    nombre = input()
    print("Introduce el numero de telefono")
    telefono = input()
    print("Introduce el correo")
    correo = input()
    # antes de hacer nada mas, lo metemos en la lista, y
sacamos la lista
    agenda.append([nombre,telefono,correo])
    print(agenda)
    # Guardo en archivo
    archivo = open("agenda.txt",'a')
    longaniza = nombre+","+telefono+","+correo+"\n"
    archivo.write(str(longaniza))
    archivo.close()
    # ejecucion recursiva
    miAgenda()

miAgenda()
```

Al ejecutar esta nueva iteración del programa dentro del terminal, podremos comprobar como cada vez que realizamos una ejecución del programa, guarda los datos recabados por parte del usuario en el archivo de texto llamado agenda txc

Esta información no saldrá necesariamente en la consola, sino que tendremos que abrir de forma manual el archivo de texto para comprobar que contiene la información que se le ha pedido anteriormente al usuario a través de la terminal

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============= RESTART:
/Users/josevicente/Desktop/Python/agenda2.py ============
Introduce el nuevo nombre en la agenda
Jose Vicente
Introduce el numero de telefono
1234
Introduce el correo
info@jocarsa.com
[['nombre', 'telefono', 'email'], ['Jose Vicente', '1234',
'info@jocarsa.com']]
Introduce el nuevo nombre en la agenda
```

10.3. Persistencia: lectura

En la tercera parte de este programa informático implementamos la recuperación de los datos previamente guardados dentro del archivo de texto.

Para ello, voy a replantear el programa desde cero, realizando un desarrollo basado en el código que teníamos en la versión anterior, pero ampliándolo y modificándolo sensiblemente Para incorporar la operación de recuperación de los datos previamente guardados.

En la primera parte del ejercicio introducimos un comentario para presentar el programa.

```
##Programa agenda por Jose Vicente Carratala


```

A continuación creamos una estructura de datos de la misma forma que lo hemos hecho anteriormente, y, por supuesto, manteniendo el mismo modelo de datos, en el sentido de que tendremos triadas de datos, que podríamos llamar tu plush, que contendrá en primer lugar el nombre, en segundo lugar el teléfono, y en tercer lugar el correo electrónico.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]
```

Una primera diferencia que podemos encontrar con respecto al desarrollo anterior es que antes de realizar ninguna otra operación, vamos a cargar el contenido del archivo de texto, en el cual hemos estado guardando registros, y para ello, en primer lugar, abrimos el archivo y lo colocamos dentro de una variable.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
archivo = open("agenda.txt",'r')
```

A continuación vamos a realizar un repaso de las líneas del archivo utilizando un bucle *for* que leerá una única las líneas, y las pondrá dentro de una variable llamada nueva línea.

Para ello debemos tener en cuenta que hemos utilizado por una parte el método *readline* para leer cada una de las líneas del archivo, pero a continuación, muy importante, utilizamos una instrucción de *Split* para tomar una cadena de caracteres, y convertirla en una lista, utilizando el carácter que nosotros designemos, que en este caso ese es el signo de la,, para elegir, cuál es el punto de corte en el cual una cadena se convierte en una lista.

Llegados a este punto, debemos tener en cuenta que si en la parte anterior del ejercicio hemos utilizado el símbolo de la , para encadenar los datos al guardarlos dentro del archivo, en este caso, en el cual estamos recuperando los datos desde el archivo de vuelta al programa, tenemos que usar exactamente el mismo carácter que hemos utilizado en el momento de guardar los datos.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
archivo = open("agenda.txt",'r')
for i in range(1,10):
    nuevalinea = archivo.readline().split(",")
```

Por último, guardamos los datos dentro de la estructura agenda que hemos creado anteriormente.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
archivo = open("agenda.txt",'r')
for i in range(1,10):
    nuevalinea = archivo.readline().split(",")
    agenda.append(nuevalinea)
```

Con esto, lo que habremos conseguido es no solo traer de vuelta los datos desde el archivo de texto, sino introducirlos dentro de la lista bidimensional a la que hemos denominado agenda, con lo cual, en la siguiente parte del programa, lo que hacemos es lanzarlo por pantalla mediante un comando

Print, simplemente para comprobar que todos los datos se han cargado correctamente desde el archivo de texto.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
archivo = open("agenda.txt",'r')
for i in range(1,10):
    nuevalinea = archivo.readline().split(",")
    agenda.append(nuevalinea)

#antes de seguir, dime en que estado esta la agenda
print(agenda)
```

Una vez que hemos realizado esta operación, a continuación creamos de nuevo la función de mi agenda, tras lo cual, otra de las novedades que introducimos en esta versión del ejercicio, consiste en presentar al usuario un menú donde podrá elegir, en este caso y para empezar, dos operaciones que serán, en primer lugar, poder insertar un registro nuevo, y en segundo lugar, poder realizar un listado de los registros.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
archivo = open("agenda.txt",'r')
for i in range(1,10):
    nuevalinea = archivo.readline().split(",")
    agenda.append(nuevalinea)
```

```
#antes de seguir, dime en que estado esta la agenda
print(agenda)

def miAgenda():
    #Menu inicial
    print("Escoge tu opcion")
    print("1.-Introduce nuevo registro")
    print("2.-Listar registros")
    print("3.-Buscar registro")
```

A partir de aquí, capturamos la entrada del usuario utilizando una función input.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
archivo = open("agenda.txt",'r')
for i in range(1,10):
    nuevalinea = archivo.readline().split(",")
    agenda.append(nuevalinea)

#antes de seguir, dime en que estado esta la agenda
print(agenda)

def miAgenda():
    #Menu inicial
    print("Escoge tu opcion")
    print("1.-Introduce nuevo registro")
    print("2.-Listar registros")
    print("3.-Buscar registro")
    opcion = input()
```

Y una vez que hemos realizado esta operación, de la misma forma que lo hiciéramos anteriormente dentro del ejercicio de la calculadora, atrapamos cada uno de los casos posibles utilizando la estructura de control *if*, para ejecutar el código correspondiente en cada uno de los dos casos que hasta el momento hemos definido.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
archivo = open("agenda.txt",'r')
for i in range(1,10):
    nuevalinea = archivo.readline().split(",")
    agenda.append(nuevalinea)

#antes de seguir, dime en que estado esta la agenda
print(agenda)

def miAgenda():
    #Menu inicial
    print("Escoge tu opcion")
    print("1.-Introduce nuevo registro")
    print("2.-Listar registros")
    print("3.-Buscar registro")
    opcion = input()
    if opcion == "1":
```

Podemos comprobar que en el primer caso, volvemos a tener un código muy similar al que teníamos anteriormente, en el cual, en el caso de querer introducir un nuevo registro, se le pregunta al usuario acerca de los datos de ese registro, y por último se inserta el registro dentro del archivo de texto.

```
##Programa agenda por Jose Vicente Carratala
```

```
agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
archivo = open("agenda.txt",'r')
for i in range(1,10):
    nuevalinea = archivo.readline().split(",")
    agenda.append(nuevalinea)

#antes de seguir, dime en que estado esta la agenda
print(agenda)

def miAgenda():
    #Menu inicial
    print("Escoge tu opcion")
    print("1.-Introduce nuevo registro")
    print("2.-Listar registros")
    print("3.-Buscar registro")
    opcion = input()
    if opcion == "1":
        print("Introduce el nuevo nombre en la agenda")
        nombre = input()
        print("Introduce el numero de telefono")
        telefono = input()
        print("Introduce el correo")
        correo = input()
        # antes de hacer nada mas, lo metemos en la lista, y
sacamos la lista
        agenda.append([nombre,telefono,correo])
        ##    print(agenda)
        # Guardo en archivo
        archivo = open("agenda.txt",'a')
        longaniza = nombre+","+telefono+","+correo+"\n"
        archivo.write(str(longaniza))
        archivo.close()
```

En el caso de que la opción seleccionada haya sido la segunda, realizar un listado de los registros de la memoria, simplemente se hace un pequeño bucle *for* que recorre la longitud de la agenda, y se presenta cada uno de los registros en pantalla.

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
archivo = open("agenda.txt",'r')
for i in range(1,10):
    nuevalinea = archivo.readline().split(",")
    agenda.append(nuevalinea)

#antes de seguir, dime en que estado esta la agenda
print(agenda)

def miAgenda():
    #Menu inicial
    print("Escoge tu opcion")
    print("1.-Introduce nuevo registro")
    print("2.-Listar registros")
    print("3.-Buscar registro")
    opcion = input()
    if opcion == "1":
        print("Introduce el nuevo nombre en la agenda")
        nombre = input()
        print("Introduce el numero de telefono")
        telefono = input()
        print("Introduce el correo")
        correo = input()
        # antes de hacer nada mas, lo metemos en la lista, y
```

```
sacamos la lista
        agenda.append([nombre,telefono,correo])
        ##    print(agenda)
        # Guardo en archivo
        archivo = open("agenda.txt",'a')
        longaniza = nombre+","+telefono+","+correo+"\n"
        archivo.write(str(longaniza))
        archivo.close()
    if opcion == "2":
        for i in range(1,len(agenda)):
            print(agenda[i])
```

Por supuesto, al final de cualquiera de los dos casos volvemos a ejecutar de forma recursiva la función, para que el programa se ejecute una y otra vez en bucle infinito,

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
archivo = open("agenda.txt",'r')
for i in range(1,10):
    nuevalinea = archivo.readline().split(",")
    agenda.append(nuevalinea)

#antes de seguir, dime en que estado esta la agenda
print(agenda)

def miAgenda():
    #Menu inicial
    print("Escoge tu opcion")
    print("1.-Introduce nuevo registro")
    print("2.-Listar registros")
```

```
    print("3.-Buscar registro")
    opcion = input()
    if opcion == "1":
        print("Introduce el nuevo nombre en la agenda")
        nombre = input()
        print("Introduce el numero de telefono")
        telefono = input()
        print("Introduce el correo")
        correo = input()
        # antes de hacer nada mas, lo metemos en la lista, y
sacamos la lista
        agenda.append([nombre,telefono,correo])
        ##    print(agenda)
        # Guardo en archivo
        archivo = open("agenda.txt",'a')
        longaniza = nombre+","+telefono+","+correo+"\n"
        archivo.write(str(longaniza))
        archivo.close()
    if opcion == "2":
        for i in range(1,len(agenda)):
            print(agenda[i])
    # ejecucion recursiva
    miAgenda()
```

Y por último ejecutamos al final del todo la función principal, para que se lance su ejecución cuando ejecutamos el programa por primera vez desde la consola

```
##Programa agenda por Jose Vicente Carratala

agenda = [["nombre","telefono","email"]]

#antes de nada, vamos a cargar los registros que teniamos en
el archivo de texto
```

```
archivo = open("agenda.txt",'r')
for i in range(1,10):
    nuevalinea = archivo.readline().split(",")
    agenda.append(nuevalinea)

#antes de seguir, dime en que estado esta la agenda
print(agenda)

def miAgenda():
    #Menu inicial
    print("Escoge tu opcion")
    print("1.-Introduce nuevo registro")
    print("2.-Listar registros")
    print("3.-Buscar registro")
    opcion = input()
    if opcion == "1":
        print("Introduce el nuevo nombre en la agenda")
        nombre = input()
        print("Introduce el numero de telefono")
        telefono = input()
        print("Introduce el correo")
        correo = input()
        # antes de hacer nada mas, lo metemos en la lista, y
sacamos la lista
        agenda.append([nombre,telefono,correo])
        ##    print(agenda)
        # Guardo en archivo
        archivo = open("agenda.txt",'a')
        longaniza = nombre+","+telefono+","+correo+"\n"
        archivo.write(str(longaniza))
        archivo.close()
    if opcion == "2":
        for i in range(1,len(agenda)):
            print(agenda[i])
    # ejecucion recursiva
    miAgenda()
```

```
miAgenda()
```

11. Programación orientada a objetos

11.1. Trabajo con clases

11.1.1. Declaración de una clase

En primer lugar, cuando trabajamos con clases y objetos, es importante crear una clase, como plantilla para continuar trabajando.

En Python, las clases se declaran utilizando la palabra reservada claras, a continuación el nombre de la clase, y a continuación:.

Una vez que hayamos hecho eso, deberemos sangrar las siguientes líneas, como cuando declaramos una función o cuando utilizamos una estructura de control del flujo de la información, para que el intérprete entienda que las líneas de código que presentamos a continuación forman parte de la clase.

Es muy recomendable que el nombre de una clase empiece con una letra mayúsculas.

En Python, a diferencia de otros lenguajes de programación, se nos permiten que los nombres de las clases empiecen con letras minúsculas, pero por una sencilla cuestión de buenas prácticas en la nomenclatura de aquellos recursos que vayamos a utilizar, se recomienda ampliamente utilizar nomenclatura mediante la cual el número de la clase empieza con una letra mayúsculas, para más adelante, poder diferenciar correctamente las variables, de los métodos, de los objetos.

```
##Esto es la definición de una clase
class Persona:
```

11.1.2. Constructor de la clase

Una vez que declaramos una clase, debemos declarar un método constructor.

En Python, el método constructor se define y se declara escribiendo dos guiones bajos, a continuación la palabra INIT, y por último dos guiones bajos más.

El método constructor es uno de los métodos que acepta una clase, y como tal, realmente es una función introducida dentro de una clase.

Es por esto que, como función que es, presenta unos paréntesis después de la palabra reservada init, el cual es deberemos introducir los parámetros que van a formar parte de ese constructor.

Para este ejercicio, en el cual estamos creando la clase que corresponde a una persona, introducimos los parámetros que decidirán a la persona, tales, como por ejemplo su nombre, su edad, su apellido, y su color del pelo.

Una vez que estamos dentro del método constructor, veremos que aparece una palabra reservada muy importante que es Cel. A continuación, veremos que igualamos los parámetros que hemos introducido dentro del método constructor a una serie de variables que tienen los mismos nombres, pero que presentan ese prefijo llamado *self*.

Ese prefijo indica que lo que estamos creando a continuación es una propiedad y no estamos llamando a una variable cualquiera que esté en uso dentro del programa.

```
##Esto es la definición de una clase
class Persona:
    def __init__(self,nombre,edad,apellido,colorpelo):
```

11.1.3. Propiedades

Las propiedades son variables que existen dentro de una clase, y que sirven para gestionar la información que utiliza esa clase.

Una clase puede utilizar variables globales que existan dentro del programa y que estén en uso en ese momento, pero tiene, en sus propiedades, la posibilidad de utilizar variables que únicamente existen dentro de esa clase.

```
##Esto es la definición de una clase
class Persona:
    def __init__(self,nombre,edad,apellido,colorpelo):
        self.nombre = nombre
        self.edad = edad
        self.apellido = apellido
        self.colorpelo = colorpelo
```

11.1.4. Métodos

Como he comentado anteriormente, si las propiedades de una clase son realmente variables, los métodos de una clase son realmente funciones.

Más allá de la función del método constructor, una clase puede presentar tantos métodos como sea necesario para que esa clase funcione correctamente y para que realice todas aquellas acciones que tenga que realizar.

En este ejemplo, creamos una función, dentro de la clase, en la cual la clase se presenta, utilizando, además, una de las propiedades que hemos definido previamente dentro de esa clase.

```
##Esto es la definición de una clase
class Persona:
    def __init__(self,nombre,edad,apellido,colorpelo):
        self.nombre = nombre
        self.edad = edad
        self.apellido = apellido
        self.colorpelo = colorpelo

    def mePresento(self):
        print("Hola, mi nombre es "+self.nombre)
```

11.1.5. Uso de la clase

Hasta el momento hemos creado una clase, pero no la hemos utilizado.

Para utilizarla, tendremos que introducir el código que se muestra a continuación, en el cual veremos que creamos a dos personas diferentes, y en cada una de las personas introducimos diferentes parámetros para establecer el nombre, la edad, el apellido, y el color del pelo de esa persona.

Deberemos tener en cuenta que estos parámetros, cuatro parámetros para este ejemplo, corresponden a los cuatro parámetros que hemos llamado anteriormente dentro del método constructor.

Si nos fijamos, sin embargo, en el método constructor, podremos comprobar cómo, con anterioridad, no habíamos indicado cuatro, sino cinco parámetros.

Nos fijamos en que el primero de los cinco parámetros era el parámetro self, que no es realmente un parámetro definido por nosotros, sino que es un parámetro cuya presencia es obligatoria, y, por lo tanto no se cuenta dentro de los cuatro parámetros que hemos utilizado para definir un nuevo objeto.

En cuanto a nomenclatura, deberemos tener en cuenta que en el momento en el que definimos esta estructura, la estructura recibe el nombre de clase, pero en el momento en el que seleccionamos esa plantilla y creamos Elementos a partir de esa plantilla, ese proceso de crear un elemento a partir de la clase se llama instanciación, y el objeto creado mediante el proceso de instanciación recibe el nombre de objeto.

Este es el motivo por el cual este paradigma de programación recibe el nombre de programación orientada a objetos, ya que el objetivo es crear una o varias plantillas en forma de clases, y luego crear, derivar, tantos objetos, como sea necesario a partir de esa clase.

```
##Esto es la definición de una clase
class Persona:
    def __init__(self,nombre,edad,apellido,colorpelo):
        self.nombre = nombre
        self.edad = edad
```

```
        self.apellido = apellido
        self.colorpelo = colorpelo

    def mePresento(self):
        print("Hola, mi nombre es "+self.nombre)

persona1 = Persona("Juan",0,"Lopez","negro")
persona2 = Persona("Jaime",3,"García","rubio")
```

Para ejecutar el código comprobaremos cómo se han creado dos instancias de la clase persona, habiéndo otorgado a cada una de esas instancias parámetros para especificar el nombre, la edad , su apellido, y el color del pelo.

Sin embargo , esto no devuelve necesariamente ningún resultado en la consola, ya que lo que hemos realizado únicamente ha consistido en introducir información dentro de la memoria del sistema

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

======== RESTART:
/Users/josevicente/Desktop/cursodepython/clasesuno.py
========
```

11.1.6. Acceso a las propiedades de la clase

Aunque no es especialmente recomendable, se puede acceder directamente a las propiedades de un objeto, una vez que ese objeto ha sido creado.

Para ello, una vez que hemos instanciado, una copia de la clase dentro de un objeto, simplemente podemos llamar a las propiedades del objeto utilizando el operador.

No solo, de hecho, podemos acceder para leer las propiedades del objeto, sino que también podemos modificar las propiedades de ese mismo objeto, simplemente con el código que se presenta a continuación.

Que se pueda realizar ese tipo de operaciones no quiere decir que sea recomendable realizarlo de esa forma. En próximos apartados dentro de este mismo capítulo, vamos a tratar esa cuestión, y vamos a presentar una forma más correcta de poder acceder a leer los parámetros de un objeto, y también a poder modificar las propiedades de ese objeto.

```
##Esto es la definición de una clase
class Persona:
    def __init__(self,nombre,edad,apellido,colorpelo):
        self.nombre = nombre
        self.edad = edad
        self.apellido = apellido
        self.colorpelo = colorpelo

    def mePresento(self):
        print("Hola, mi nombre es "+self.nombre)
```

Al ejecutar este código podemos comprobar cómo, en primer lugar, declaramos la plantilla de una clase que se guarda en memoria.

Si ejecutamos este código el sistema no devolverá ningún resultado en la terminal, ya que únicamente hemos declarado la plantilla de la clase de una

persona, pero en ningún momento estamos utilizando todo, al menos todavía, dicha plantilla de la clase.

```
##Esto es la definición de una clase
class Persona:
    def __init__(self,nombre,edad,apellido,colorpelo):
        self.nombre = nombre
        self.edad = edad
        self.apellido = apellido
        self.colorpelo = colorpelo

    def mePresento(self):
        print("Hola, mi nombre es "+self.nombre)

persona1 = Persona("Juan",0,"Lopez","negro")
persona2 = Persona("Jaime",3,"García","rubio")
```

A continuación, en esta siguiente parte del ejercicio, declaramos dos nuevas instancias de la clase en dos entidades que, a partir de ese momento, pasan a denominarse objetos, o instancias de la clase.

cada una de las instancias recibe automáticamente todas las propiedades de la clase, al igual que todos los métodos que la clase contuviera.

Además, teniendo en cuenta de que en el método constructor hemos especificado que vamos a llamar a cuatro parámetros, que son el nombre, la edad, el apellido y el color del pelo, al instanciar los objetos tenemos que llamar a esos cuatro parámetros dentro del constructor.

Sin embargo, si ejecutamos este código todavía no habrá ningún resultado en la terminal.

```
##Esto es la definición de una clase
class Persona:
    def __init__(self,nombre,edad,apellido,colorpelo):
        self.nombre = nombre
        self.edad = edad
        self.apellido = apellido
        self.colorpelo = colorpelo

    def mePresento(self):
        print("Hola, mi nombre es "+self.nombre)

persona1 = Persona("Juan",0,"Lopez","negro")
persona2 = Persona("Jaime",3,"García","rubio")


print(persona1.nombre)
print(persona2.nombre)
```

Sólo cuando cumplimos la condición de no solo declarar la clase, ni tampoco tampoco instanciar los objetos, sino llamar a un método de la clase, o llamar directamente a una propiedad, es cuando entonces podemos ver algo de resultado en la consola.

En este caso, y para este ejemplo, una vez que hemos declarado dos objetos, a continuación lanzamos por pantalla cada uno de los nombres de las dos instancias, para comprobar que cada una de las instancias realmente es una burbuja de información completamente independiente al resto de instancias.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.
```

```
===== RESTART:
/Users/josevicente/Desktop/Python/accesopropiedadesclase.py
=====
Juan
Jaime
```

11.1.7. Llamada a los métodos de la clase

Anteriormente habíamos definido los métodos de una clase, y a continuación vamos a mostrar cómo realizar llamadas a los métodos de esa clase.

En este caso no solo realizamos una llamada al método declarado anteriormente, sino que además podemos observar, al realizar una impresión en la consola, cuál es el sentido de que cada uno de los objetos tenga sus propias propiedades, y es que, como podemos comprobar, cada objeto, se convierte en una burbuja que maneja sus propias propiedades en forma de variables, y las variables no se mezclan entre diferentes objetos, resultando una estructura muy conveniente cuando necesitamos trabajar con piezas Separadas de información.

```
##Esto es la definición de una clase
class Persona:
    def __init__(self,nombre,edad,apellido,colorpelo):
        self.nombre = nombre
        self.edad = edad
        self.apellido = apellido
        self.colorpelo = colorpelo

    def mePresento(self):
        print("Hola, mi nombre es "+self.nombre)

persona1 = Persona("Juan",0,"Lopez","negro")
persona2 = Persona("Jaime",3,"García","rubio")
```

```
print(persona1.nombre)
print(persona2.nombre)

persona1.mePresento()
persona1.nombre = "Jorge"
persona1.mePresento()

del persona1

persona1.mePresento()
```

Al finalizar la ejecución del programa, podremos comprobar cómo no solo somos capaces de crear instancias de objetos, sino que , al llamar a los métodos del objeto, así como a sus propiedades, podemos obtener resultados impresos por pantalla.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

======= RESTART:
/Users/josevicente/Desktop/Python/llamadametodosclase.py
======
Juan
Jaime
Hola, mi nombre es Juan
Hola, mi nombre es Jorge
```

11.2. Eliminación de un objeto

La gestión de la memoria es uno de los grandes retos de prácticamente cualquier programa informático. En este caso, podemos comprobar como tenemos la instrucción de eliminar, que nos permite borrar de la memoria las instancias que ya no vayamos a utilizar.

```
##Esto es la definición de una clase
class Persona:
    def __init__(self,nombre,edad,apellido,colorpelo):
        self.nombre = nombre
        self.edad = edad
        self.apellido = apellido
        self.colorpelo = colorpelo

    def mePresento(self):
        print("Hola, mi nombre es "+self.nombre)

persona1 = Persona("Juan",0,"Lopez","negro")
persona2 = Persona("Jaime",3,"García","rubio")


print(persona1.nombre)
print(persona2.nombre)

persona1.mePresento()
persona1.nombre = "Jorge"
persona1.mePresento()

del persona1

persona1.mePresento()
```

durante la ejecución del ejercicio anterior vamos a poder comprobar cómo creamos una instancia de la clase, utilizamos uno de sus métodos, a continuación eliminamos la instancia, y al intentar ejecutar uno de los métodos de instancia, el intérprete da error, puesto que esa instancia ha sido borrada de la memoria del ordenador, y por tanto ya no es accesible.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

======== RESTART:
/Users/josevicente/Desktop/Python/eliminacionobjeto.py
=======
Juan
Jaime
Hola, mi nombre es Juan
Hola, mi nombre es Jorge
Traceback (most recent call last):
  File
"/Users/josevicente/Desktop/Python/eliminacionobjeto.py",
line 25, in <module>
    persona1.mePresento()
NameError: name 'persona1' is not defined. Did you mean:
'persona2'?
```

11.3. Herencia de clases

El trabajo con clases presenta diferentes herramientas, y dentro de ellas, en este capítulo, presentamos un concepto que es la herencia.

Mediante la herencia podemos tener jerarquías de clases, siendo unas de ellas las clases parentales, o clases madres, y otras de ellas, las clases filiales, o clases hijas.

Para ilustrar este ejemplo, hemos creado un trozo de código en el cual creamos tres clases iniciales, con un ejemplo que suele ser muy frecuente

cuando se explica la programación orientada objetos dentro de cualquier lenguaje de programación, que consiste en elevar diferentes categorías del reino animal.

Así que empezamos con una clase llamada gato, que tiene una serie de parámetros y tiene una serie de métodos.

```
class Gato():
    def __init__(self,color):
        self.color = color

    def maulla(self):
        print("El gato está maullando")

micifu = Gato("naranja")
micifu.maulla()
```

Por ejemplo, el gato tiene la capacidad concreta de poder maullar.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============ RESTART:
/Users/josevicente/Desktop/Python/clasegato.py ===========
El gato está maullando
```

Sin embargo, el gato, forma parte del género mamífero, al igual que otros muchos animales.

Los mamíferos se distinguen por tener la capacidad de mamar cuando nacen, y es por esto que, a continuación, vamos a indicarle a la clase gato que deriva de la clase mamífero.

```
class Mamifero():
    def __init__(self,edad):
        self.edad = edad
    def mama(self):
        print("este animal mama al nacer")

class Gato(Mamifero):
    def __init__(self,color):
        self.color = color

    def maulla(self):
        print("El gato esta maullando")

micifu = Gato("naranja")
micifu.maulla()

micifu.mama()
```

Cuando realizamos esta indicación, automáticamente la clase, gato, hereda todas las propiedades y todos los métodos que tuviera la clase Manicero.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============ RESTART:
/Users/josevicente/Desktop/Python/herencia1.py ===========
El gato esta maullando
este animal mama al nacer
```

Pero no solo podemos realizar una herencia de un solo nivel jerárquico, sino que podemos utilizar tantos niveles como queramos.

En el siguiente ejercicio, tenemos una clase animal, ya qué los mamíferos forman parte del reino animal.

La clase animal presenta una serie de propiedades y una serie de métodos, y en el momento en el que la clase mamífero hereda de la clase animal, automáticamente la clase mamífero, obtiene todos los parámetros y todos los métodos de la clase animal, pero no solo eso, sino que además deberemos tener en cuenta que la clase de gato hereda no sólo todo lo que tiene la clase mamífero, sino a su vez, en cascada, todo aquello que tiene la clase animal.

```
class Animal:
    def __init__(self,altura):
        self.altura = altura
    def salta(self):
        print("este animal es capaz de saltar")

class Mamifero(Animal):
    def __init__(self,edad):
        self.edad = edad
    def mama(self):
        print("este animal mama al nacer")

class Gato(Mamifero):
    def __init__(self,color):
        self.color = color

    def maulla(self):
        print("El gato esta maullando")

micifu = Gato("naranja")
micifu.maulla()

micifu.mama()
micifu.salta()
```

Al realizar una ejecución del código obtendremos el siguiente resultado en la

consola, en el cual podremos comprobar que , al llamar a los métodos de la clase , estos realizan una impresión por pantalla.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
/Users/josevicente/Desktop/Python/herencia2.py ===========
El gato esta maullando
este animal mama al nacer
este animal es capaz de saltar
```

11.4. Propiedades privadas / Setters y getters

Dentro del concepto de encapsulación presente en prácticamente cualquier lenguaje de programación, encontramos que , cuando declaramos una clase, en cierta forma queremos tener un control lo más estricto posible de la visibilidad de los datos que contiene esa clase, y por tanto sus instancias, al igual que queremos controlar el tipo de acceso que los usuarios, es decir como los programadores, tendrán con respecto de esa información.

es por esto que en el siguiente ejemplo declaramos una variable llamada saldo, pero a diferencia de lo que hemos hecho hasta el momento, dicha variable viene precedida por la palabra reservada private.

es la palabra indica que esa propiedad existe dentro de la clase, pero que, sin embargo, no es visible, ni para leer ni para escribir, desde fuera de la clase.

Si la propiedad no es accesible desde fuera de la clase de un modo directo, en ese caso es cuando tenemos que programar dos métodos, que suelen recibir el nombre de setters, y getters, es decir, son métodos encargados de cambiar el valor de la variable, en el caso de los setters, iré obtener el valor de la variable en el caso de los getters.

```
class CuentaBancaria:
    def __init__(self,numero,nombre,saldo):
        self.numero = numero
        self.nombre = nombre
        private self.saldo = saldo
    def modificaSaldo(nuevosaldo):
        self.saldo = nuevosaldo

cuenta1 = CuentaBancaria("0001","jose Vicente",1000)

print(cuenta1.saldo)
cuenta1.saldo = 1000000000000000
print(cuenta1.saldo)
```

12. Desarrollo de interfaces gráficas

12.1. Introducción

A continuación, en este apartado, vamos a presentar el concepto de desarrollo de interfaces gráficas para los programas que estamos creando en Python.

Dentro de lo que es el desarrollo de interfaces gráficas de usuario, existen múltiples librerías que podemos utilizar con el lenguaje de programación Python, pero sin duda, la librería gráfica más utilizada, que además viene pre instalada con la distribución estándar de Python, *TKInter*.

Gracias a esta librería vamos a poder crear, de una forma tremendamente sencilla, aplicaciones en ventanas que mostrarán a los usuarios atractivas, interfaces gráficas con las cuales podrán interactuar.

Por lo tanto, hasta este punto de la publicación, hemos trabajado creando aplicaciones en consola, es decir, aplicaciones que presentaban información en modo texto en una consola o terminal, y que requerían información al usuario utilizando la misma consola.

Sin embargo, desde hace ya muchos años, los sistemas operativos no son de línea de comando, sino que presentan interfaces gráficas basadas en ventanas.

Es por esto que esta librería, al igual que cualquier otra librería de desarrollo de interfaz de usuario gráficas en ventanas, nos permite crear aplicaciones que muestran este tipo de recurso gráfico, y con las cuales el usuario puede acceder de una forma mucho más cómoda a la interacción con la aplicación de software.

Dicho esto, muchas veces, a lo largo de los años, me he encontrado que los alumnos, cuando llegan a esta parte del contenido, es cuando dicen " ahora es cuando voy a realizar realmente programas informáticos ".

Realizar programas informáticos, realmente, es lo que hemos hecho hasta el momento en los capítulos anteriores del libro.

Lo que vamos hacer a continuación, no es generar programas, sino generar interfaz de usuario gráficas, y en ventanas, para dichos programas.

12.2. Creación de etiquetas

Y vamos a crear un primer programa utilizando la librería *TKInter*, y para ello, en primer lugar, en la primera línea, comprobamos que tengamos la librería correctamente instalada dentro del sistema.

```
from tkinter import *
```

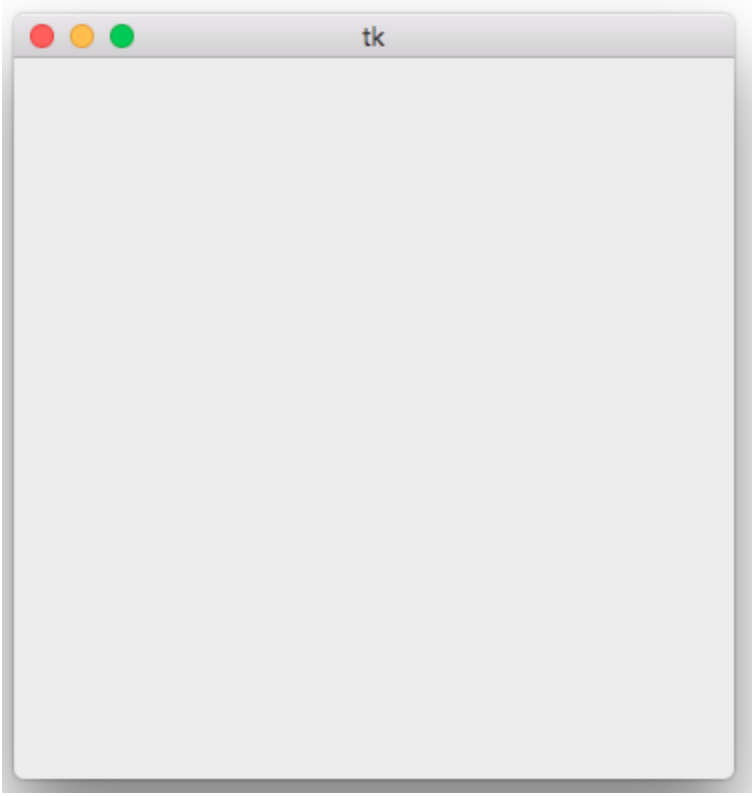
A continuación, creamos un primer recurso dentro de nuestro programa, que es un Frame, o un Marco. Un Frame consiste en un cuadrado dentro del cual podemos introducir diferentes elementos, componentes, o widgets de interfaz de usuario.

Para este ejemplo hemos aprovechado que creamos un Frame para introducir dos parámetros, como son la altura y la anchura.

En este ejemplo creamos una ventana cuadrada que tiene 300px de ancho por 300px de alto.

```
from tkinter import *

f = Frame(width=300,height=300)
f.pack(padx=30,pady=30)
```

Deberemos tener en cuenta que la medida de proporciones dentro de esta librería se expresa en píxeles por defecto, aunque al introducir el parámetro, introducimos un número entero sin unidades.

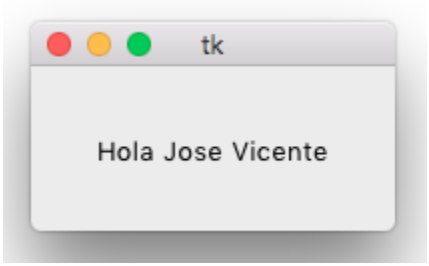


A continuación, empaquetamos el recurso que acabamos de definir para que el programa realmente ejecute el hecho de mostrar ese recurso dentro de la ventana que vamos a crear.

Debemos tener en cuenta que crear el recurso significa almacenarlo en la memoria, pero no necesariamente mostrarlo en pantalla. Hasta que no utilizamos el comando *pack*, no es realmente hasta ese punto en el que el programa entiende que debe mostrar el recurso dentro de la ventana gráfica que estamos creando.

Una vez que hemos creado esta ventana, a continuación vamos a crear un elemento gráfico inicial, que es un label. Los label son etiquetas que nos permiten introducir texto dentro de una ventana gráfica, y requieren llamar al comando empaquetar de la misma forma que lo hemos llamado en el ejemplo anterior.

```
from tkinter import *

f = Frame(width=300,height=300)
f.pack(padx=30,pady=30)

titulo = Label(f,text = "Hola Jose Vicente")
titulo.pack(side=TOP)
```



12.3. Creacion de etiquetas avanzadas e imágenes

En el siguiente ejemplo, vamos a profundizar en el uso de la librería de creación de interfaces gráficas, mediante el añadido de nuevos elementos, y mediante la revisión de nuevos parámetros para aquellos elementos que ya conocíamos de forma previa.

En este ejercicio vamos a empezar importando la librería

```
from tkinter import *

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

titulo = Label(marco,text="Programa agenda
v0.1",fg="black",font=("Arial,Verdana,sans-serif",24))
titulo.pack(side=TOP)
```

```
autor = Label(marco,text="Jose Vicente
Carratala",fg="grey",font=("Arial,Verdana,sans-serif",16))
autor.pack(side=TOP)

foto =
PhotoImage(file="josevicente.jpg",width=100,height=100)

textofoto = Label(marco,image=foto)
textofoto.pack(side=TOP)

mainloop()
```

Y a continuación generaremos un Marco, y añadiremos el Marco a la ventana principal utilizando el comando *pack*.

```
from tkinter import *

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

titulo = Label(marco,text="Programa agenda
v0.1",fg="black",font=("Arial,Verdana,sans-serif",24))
titulo.pack(side=TOP)

autor = Label(marco,text="Jose Vicente
Carratala",fg="grey",font=("Arial,Verdana,sans-serif",16))
autor.pack(side=TOP)

foto =
PhotoImage(file="josevicente4.png",width=100,height=100)

textofoto = Label(marco,image=foto)
textofoto.pack(side=TOP)

mainloop()
```

A continuación vamos a crear una etiqueta, un label tal y como hemos visto anteriormente, la diferencia en este caso es que vamos a añadirle una serie de parámetros nuevos, Tales como por ejemplo:

El parámetro FG, que corresponde a la propiedad del *Foreground*, es decir, color de frente, a la que le asignamos una cadena que corresponde al nombre del color en inglés, en este caso Black.

El parámetro *FONT*, dentro del cual podemos introducir una tabla de parámetros, indicando por una parte, la familia de fuentes tipográficas por orden de prioridad, y en segundo lugar el tamaño de la letra.

En cuanto a la fuente deberemos tener en cuenta que en el ejemplo que estoy ilustrando, he introducido tres fuentes para que el sistema en primer lugar busque la primera de las fuentes, si falla al encontrarla, busque la segunda, y si falla al encontrarla, en ese caso, seleccione la fuente más apropiada dentro de la familia de tipo *Sans Serif*.

```
from tkinter import *

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

titulo = Label(marco,text="Programa agenda
v0.1",fg="black",font=("Arial,Verdana,sans-serif",24))
titulo.pack(side=TOP)

autor = Label(marco,text="Jose Vicente
Carratala",fg="grey",font=("Arial,Verdana,sans-serif",16))
autor.pack(side=TOP)

foto =
PhotoImage(file="josevicente4.png",width=100,height=100)
```

```
textofoto = Label(marco,image=foto)
textofoto.pack(side=TOP)

mainloop()
```

A continuación introduzco una segunda etiqueta, utilizando las mismas propiedades que hemos visto anteriormente para la primera etiqueta, en este caso, cambiando el tamaño del texto, y el color.

```
from tkinter import *

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

titulo = Label(marco,text="Programa agenda
v0.1",fg="black",font=("Arial,Verdana,sans-serif",24))
titulo.pack(side=TOP)

autor = Label(marco,text="Jose Vicente
Carratala",fg="grey",font=("Arial,Verdana,sans-serif",16))
autor.pack(side=TOP)

foto =
PhotoImage(file="josevicente4.png",width=100,height=100)

textofoto = Label(marco,image=foto)
textofoto.pack(side=TOP)

mainloop()
```

Vemos también que es posible introducir Fotografías, es decir, imágenes, insertadas dentro de las etiquetas de la interfaz gráfica de usuario.

Para ello creamos una variable llamada foto, en la cual cargamos un elemento de tipo *PhotoImage*, que admite, en este ejemplo, tres parámetros.

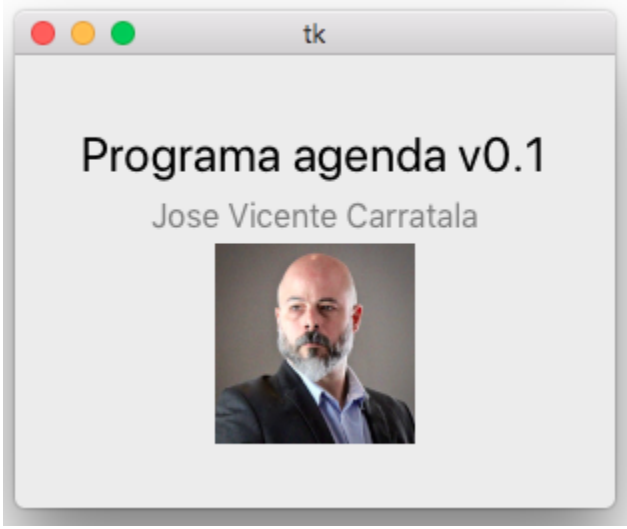
- En primer lugar fácil que nos permite especificar cuál es el archivo que vamos a cargar, debiendo estar el archivo en la misma carpeta en la que encontramos el código de Python.
- En segundo lugar la anchura medida en píxeles.
- En tercer lugar la altura medido en píxeles.

```
from tkinter import *

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

titulo = Label(marco,text="Programa agenda
v0.1",fg="black",font=("Arial,Verdana,sans-serif",24))
titulo.pack(side=TOP)

autor = Label(marco,text="Jose Vicente
Carratala",fg="grey",font=("Arial,Verdana,sans-serif",16))
autor.pack(side=TOP)

foto =
PhotoImage(file="josevicente4.png",width=100,height=100)

textofoto = Label(marco,image=foto)
textofoto.pack(side=TOP)

mainloop()
```

Éstos pasos nos han servido únicamente para cargar una imagen de memoria, así que a continuación creamos un nuevo label, es decir, una nueva etiqueta, pero en este caso comprobamos cómo en lugar de introducir contenido en texto, podemos introducir la imagen que acabamos de introducir dentro de la memoria.

```
from tkinter import *

marco = Frame(width=300,height=300)
```

```
marco.pack(padx=30,pady=30)

titulo = Label(marco,text="Programa agenda
v0.1",fg="black",font=("Arial,Verdana,sans-serif",24))
titulo.pack(side=TOP)

autor = Label(marco,text="Jose Vicente
Carratala",fg="grey",font=("Arial,Verdana,sans-serif",16))
autor.pack(side=TOP)

foto =
PhotoImage(file="josevicente4.png",width=100,height=100)

textofoto = Label(marco,image=foto)
textofoto.pack(side=TOP)

mainloop()
```



12.3.1. Consideraciones importantes en este ejercicio:

12.3.1.1. Con respecto al formato de la imagen:

Si bien la gran mayoría de imágenes que podemos encontrar en internet y a nuestro alrededor están guardadas en el formato jpg, está librería gráfica presenta serios problemas de compatibilidad con este formato, con lo cual, se recomienda que la imagen que carguemos dentro de nuestra interfaz de usuario esté en formato png.

cualquier sistema operativo razonablemente actualizado dispone de herramientas , tales como por ejemplo la vista previa en Mac, o el visor de imágenes en Windows, que nos permiten cargar imágenes en un formato, en este caso en jpg, y cambiarlas de formato, en este caso a formato PNG

12.3.1.2. Con respecto al tamaño de recorte:

Otro de los puntos débiles descargar imágenes directamente en la interfaz de usuario, al menos con el comando que lo estamos haciendo en este momento, consiste en la imposibilidad de poder reescalar la imagen, con respecto a su tamaño original, para poder controlar el tamaño en el que queremos que aparezca en pantalla.

por este motivo, deberemos introducir la imagen en el tamaño correcto dentro del programa, para lo cual, de la misma forma que se ha comentado en la sección anterior, podemos utilizar sencillas aplicaciones de edición de imágenes que se encuentren presentes dentro del sistema operativo para realizar está operación de escalado de la imagen.

por supuesto, también podemos utilizar herramientas más avanzadas de diseño gráfico y tratamiento de imágenes digitales, para realizar la operación de escalada.

12.3.1.3. Con respecto a la ubicación de la imagen

Siempre que realizó este ejercicio, suelo encontrarme con problemas con los alumnos consistentes en que la imagen no ha sido colocada en la misma

carpeta en la que se encuentra el archivo de Python con el que estamos trabajando.

Por lo tanto, es importante que nos aseguremos que tanto el archivo de Python que contiene el código fuente con el que estamos trabajando Como la imagen que vamos a utilizar, se encuentren dentro de la misma carpeta.

Eventualmente sería posible introducir las imágenes dentro de una carpeta, y llamar a la imagen formateando la carpeta y la ruta, utilizando el carácter de la barra como separador, para poder especificar ubicaciones más complejas al llamar a las imágenes.

Sin embargo, para la realización de un primer ejercicio es más conveniente que tanto el archivo Python como la imagen se encuentren dentro de la misma carpeta, para asegurar que el actual código funciona

12.4. Creación de botones

A continuación, vamos a proceder a insertar un botón, siendo el botón uno de los elementos más especiales, cuando creamos una interfaz gráfica de usuario en Python o en cualquier otro lenguaje, ya que es un elemento que, finalmente, requiere de la infracción por parte del usuario.

De la misma forma que una etiqueta label o una fotografía. No requieren interacción porque son elementos que muestran información al usuario, el botón requiere ser pulsado por el usuario en una serie de condiciones.

En este ejemplo continuamos trabajando sobre el código desarrollado en el ejercicio anterior, y en esta ocasión, al principio del código, introducimos una función que será ejecutada cuando pulsamos el botón.

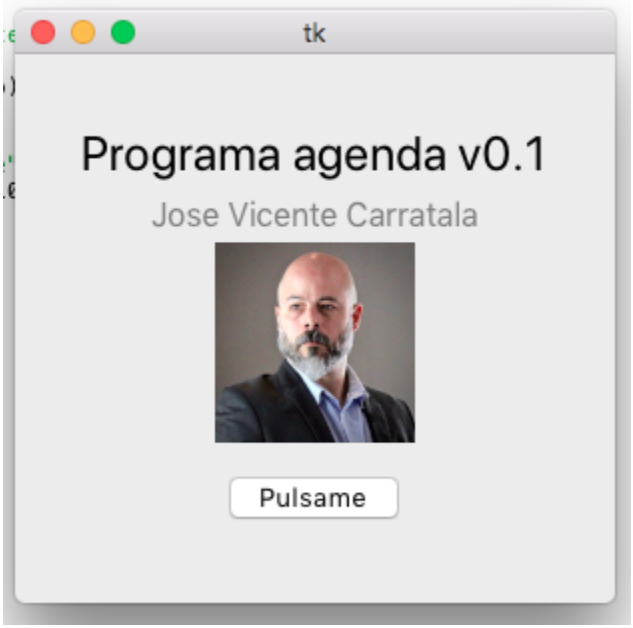
Una vez que hemos introducido la función, a continuación, al final del código, creamos un botón, que en principio tiene tres parámetros.

En primer lugar, la entidad dentro de la cual se introduce el botón, que en este caso es la variable Marco.

El texto que va a contener el botón.

Y el comando que va a ejecutar el botón, que en este caso corresponde al mismo nombre que le hemos dado a la función, para que el intérprete sepa enlazar correctamente el código que se debe ejecutar en el momento en el que se pulse el botón

```
from tkinter import *


def saluda():
    print("Has pulsado un boton")


marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

titulo = Label(marco,text="Programa agenda
v0.1",fg="black",font=("Arial,Verdana,sans-serif",24))
titulo.pack(side=TOP)

autor = Label(marco,text="Jose Vicente
Carratala",fg="grey",font=("Arial,Verdana,sans-serif",16))
autor.pack(side=TOP)

foto =
PhotoImage(file="josevicente.png",width=100,height=100)

textofoto = Label(marco,image=foto)
textofoto.pack(side=TOP)

boton = Button(marco,text="Pulsame",command=saluda)
boton.pack(side=TOP,padx=10,pady=10)


mainloop()
```



Al ejecutar el programa, podemos comprobar cómo surge una ventana flotante que tiene toda la información y todos los parámetros que hemos puesto hasta el momento en nuestro código fuente

12.5. Creación de una agenda en TKinter y SQLite

```
from tkinter import *
import sqlite3 as lite
import sys

def guarda(nombre,telefono,email):
    conexion = lite.connect("agenda.sqlite")
    cursor = conexion.cursor()
    cursor.execute("INSERT INTO contactos
VALUES(NULL,'Jorge','222222','jorge@correo.com');")
    cursor.execute("INSERT INTO contactos
```

```
VALUES(NULL,'"+nombre+"','"+telefono+"','"+email+"');")
    conexion.commit()

def lee():
    print("voy a leer la base de datos")

    conexion = lite.connect("agenda.sqlite")
    cursor = conexion.cursor()
    cursor.execute("SELECT * FROM contactos;")
    longaniza = ""
    datos = cursor.fetchall()
    for i in datos:
        longaniza += str(" nombre:"+i[1]+"\t telefono:
"+i[2]+"\t email:"+i[3]+"\n")
    titulodevuelve.insert(INSERT,longaniza)

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

titulo = Label(marco,text="Programa agenda
v0.1",fg="black",font=("Arial,Verdana,sans-serif",24))
titulo.pack(side=TOP)
autor = Label(marco,text="Jose Vicente
Carratala",fg="grey",font=("Arial,Verdana,sans-serif",16))
autor.pack(side=TOP)

foto =
PhotoImage(file="josevicente4.png",width=100,height=100)
textofoto = Label(marco,image=foto)
textofoto.pack(side=TOP)

titulo = Label(marco,text="Introduce un
nombre",fg="black",font=("Arial,Verdana,sans-serif",14))
titulo.pack(side=TOP)
camponombre = Entry(marco)
camponombre.pack(side=TOP)
```

```
titulo = Label(marco,text="Introduce un
telefono",fg="black",font=("Arial,Verdana,sans-serif",14))
titulo.pack(side=TOP)
campotelefono = Entry(marco)
campotelefono.pack(side=TOP)

titulo = Label(marco,text="Introduce un
email",fg="black",font=("Arial,Verdana,sans-serif",14))
titulo.pack(side=TOP)
campoemail = Entry(marco)
campoemail.pack(side=TOP)

boton = Button(marco,text="Guarda este registro en la base de
datos",command=lambda:guarda(camponombre.get(),campotelefono.
get(),campoemail.get()))
boton.pack(side=TOP,padx=10,pady=10)

titulo = Label(marco,text="Dame los resultados de la base de
datos",fg="black",font=("Arial,Verdana,sans-serif",14))
titulo.pack(side=TOP)

botondame = Button(marco,text="Devuelve los
registros",command=lambda:lee())
botondame.pack(side=TOP,padx=10,pady=10)

titulodevuelve = Text(marco,height=30, width=60)
titulodevuelve.pack(side=TOP)

mainloop()
```

12.6. Dibujo en Canvas

12.6.1. Dibujar

Uno de los usos más frecuentes de la creación de interfaces de usuario consiste en poder dibujar gráficos personalizados.

A continuación vamos a mostrar una serie de ejercicios en los cuales podremos comprobar que existe un objeto de tipo lienzo, gambas, dentro del cual nos es posible realizar dibujos personalizados que representan todo aquello que nuestros programas necesiten.

En este primer ejemplo, en primer lugar, vamos a comenzar importando todo lo que tiene la librería teca Inter.

A continuación crearemos un Marco que tendrá 300 px de ancho y 300 px de alto, y luego añadiremos a la aplicación.

A continuación creamos un lienzo y le decimos que es un objeto de tipo Canvas, con lo cual hereda todos los parámetros y todos los métodos que tenga el objeto indica interés.

Y a continuación vamos a realizar dos tipos de dibujo, utilizando el comando de crear línea.

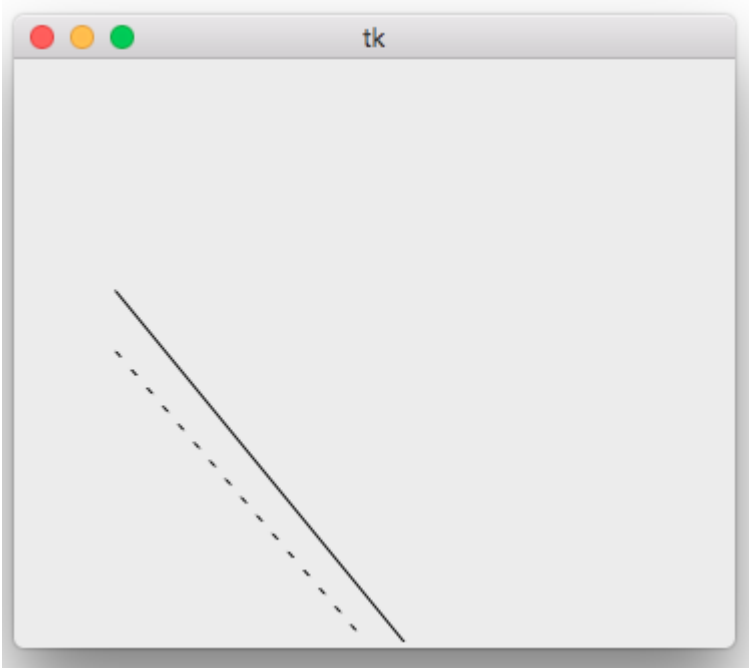
En primer lugar Creamos una línea indicando cuatro parámetros, siendo:

- El primero representa la coordenada X del punto de inicio
- Ese mundo representa la coordenada Y del punto de inicio
- El tercero representa la coordenada X del punto final
- El cuarto representa la coordenada Y del punto final

Por último creamos una línea nueva, duplicando ligeramente la anterior, para demostrar que podemos añadir parámetros adicionales, tales como por ejemplo, aquellos que nos permiten controlar el grosor o el tipo de línea.

```
from tkinter import *
```

```
marco = Frame(width=300,height=30)
marco.pack(padx=30,pady=30)

lienzo = Canvas()
lienzo.create_line(15,25,200,250)
lienzo.create_line(15,55,200,270,dash=(4,8))


lienzo.pack(side=TOP)
```



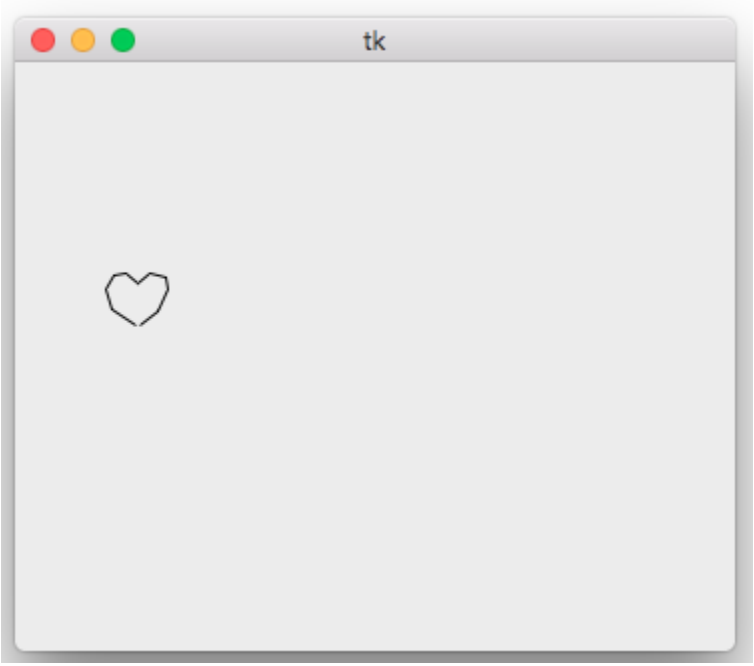
12.6.2. Creación de polilíneas

Cuando creamos una línea, realmente podemos convertirla en una poli línea simplemente asignándole parejas de nuevos parámetros.

En cada una de las parejas de parámetros que le asignamos, el primer elemento de la pareja será la componente X del nuevo punto, y el segundo parámetro de la pareja será la componente Y de ese mismo nuevo..

Por lo tanto, en este ejemplo podemos comprobar como podemos crear polilíneas complejas o simplemente añadiendo tantos parámetros como sea necesario.

```
from tkinter import *

marco = Frame(width=300,height=30)
marco.pack(padx=30,pady=30)

lienzo = Canvas()
lienzo.create_line(25.229911, 41.015623,12.756697,
32.700147,10.110863, 23.250742,13.796131,
15.880207,19.654762, 14.651784,25.985863,
19.659969,31.655506, 15.313243,39.781994,
17.203124,41.388393, 23.250743,36.474702,
33.834076,27.214286, 40.826636)



lienzo.pack(side=TOP)
```



12.7. Creación de gráficas

12.7.1. Gráfica

Uno de los usos más comunes de la librería gráfica que nos permite dibujar cosas dentro de un lienzo en Python. Consiste en poder representar visualmente piezas de información.

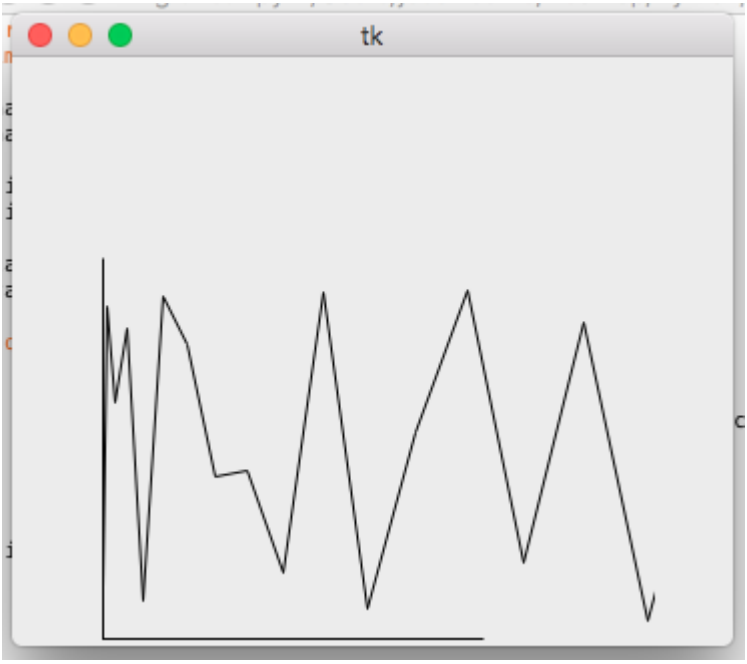
Es por esto que podemos crear gráficas de una forma muy sencilla utilizando las capacidades de pintura dentro de un lienzo de la librería de TKinter.

Es importante notar como dentro de esta misma publicación. Vamos a estar mostrando el funcionamiento de una librería estándar de Python, como es *matplotlib*, para poder generar gráficas de una forma altamente conveniente.

El hecho de mostrarlo dentro de este ejercicio nos sirve para poder ilustrar, y para que el lector comprenda el proceso de generar gráficas completamente desde cero sin tener que utilizar librerías avanzadas.

Para este ejemplo, en primer lugar, importamos las librerías necesarias, que es, en primer lugar, la librería para poder generar interfaces de usuario, y en segundo lugar a una librería de generación de números aleatorios que va a ser necesaria para el funcionamiento de este ejercicio concreto.

```
from tkinter import *
import random

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

lienzo = Canvas()
lienzo.create_line(10,10,10,200,200,200)

xanterior = 10
yanterior = 200

for i in range(0,30):
    xactual = xanterior + i*2
    yactual = random.randint(10,200)
    lienzo.create_line(xanterior,yanterior,xactual,yactual)
    xanterior = xactual
    yanterior = yactual


lienzo.pack(side=TOP)
```



A continuación creamos un Marco con una serie de medidas iniciales, y lo añadimos a la aplicación para que salga correctamente en pantalla.

```
from tkinter import *
import random

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

lienzo = Canvas()
lienzo.create_line(10,10,10,200,200,200)

xanterior = 10
yanterior = 200
```

```
for i in range(0,30):
    xactual = xanterior + i*2
    yactual = random.randint(10,200)
    lienzo.create_line(xanterior,yanterior,xactual,yactual)
    xanterior = xactual
    yanterior = yactual


lienzo.pack(side=TOP)
```

En el siguiente paso, creamos un lienzo en el que poder dibujar, y mediante aquello que hemos aprendido en los ejercicios anteriores, creamos un par de ejes mediante una polilínea que tiene tres puntos, y por lo tanto, presenta tres parejas de parámetros

```
from tkinter import *
import random

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

lienzo = Canvas()
lienzo.create_line(10,10,10,200,200,200)

xanterior = 10
yanterior = 200

for i in range(0,30):
    xactual = xanterior + i*2
    yactual = random.randint(10,200)
    lienzo.create_line(xanterior,yanterior,xactual,yactual)
    xanterior = xactual
    yanterior = yactual


lienzo.pack(side=TOP)
```

A continuación vamos a crear un punto de inicio, añadiendo un par de variables temporales, a las cuales les asignamos un valor inicial.

```
from tkinter import *
import random

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)

lienzo = Canvas()
lienzo.create_line(10,10,10,200,200,200)

xanterior = 10
yanterior = 200

for i in range(0,30):
    xactual = xanterior + i*2
    yactual = random.randint(10,200)
    lienzo.create_line(xanterior,yanterior,xactual,yactual)
    xanterior = xactual
    yanterior = yactual


lienzo.pack(side=TOP)
```

En la siguiente parte del programa, creamos un bucle de repetición foro que se ejecute 30 veces, y dentro del bucle introducimos el siguiente código para que el programa vaya trazando una serie de puntos aleatorios, que definan una gráfica.

```
from tkinter import *
import random

marco = Frame(width=300,height=300)
marco.pack(padx=30,pady=30)
```

```
lienzo = Canvas()
lienzo.create_line(10,10,10,200,200,200)

xanterior = 10
yanterior = 200

for i in range(0,30):
    xactual = xanterior + i*2
    yactual = random.randint(10,200)
    lienzo.create_line(xanterior,yanterior,xactual,yactual)
    xanterior = xactual
    yanterior = yactual


lienzo.pack(side=TOP)
```

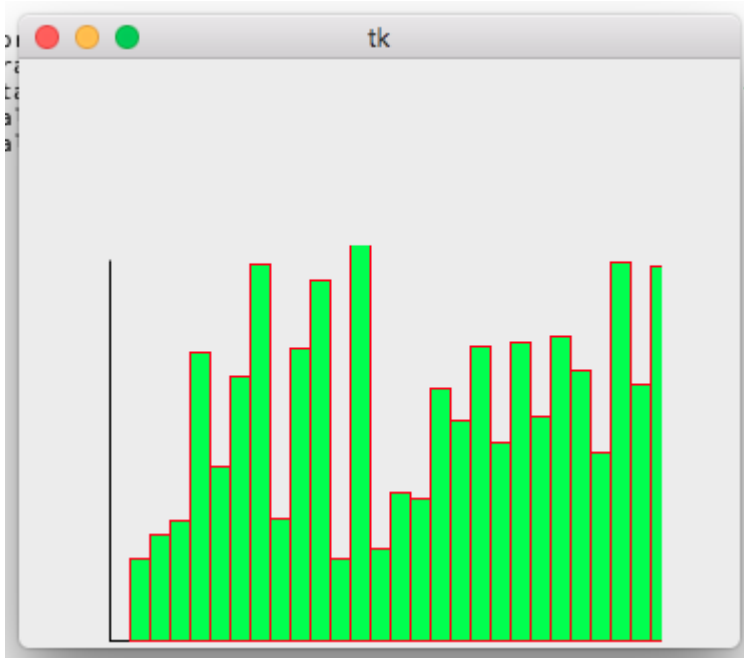
12.7.2. Gráfica barras

De la misma forma que en el ejercicio anterior, hemos creado una gráfica de líneas, a continuación vamos a modificar ligeramente el código realizado el ejercicio mediante el cual hemos generado una gráfica lineal, para crear una gráfica de barras.

Podremos comprobar que mantenemos el código del ejercicio anterior, prácticamente intacto, y únicamente dentro de la estructura de control repetitiva FORT vamos a sustituir la creación de una línea por la creación de un comando, al cual le pasaremos los mismos comandos prácticamente que hemos pasado anteriormente a la línea.

```
from tkinter import *
import random

marco = Frame(width=300,height=30)
marco.pack(padx=30,pady=30)

lienzo = Canvas()
```

```
lienzo.create_line(10,10,10,200,200,200)

xanterior = 10
yanterior = 200

for i in range(0,30):
    xactual = xanterior + 10
    yactual = random.randint(10,200)

lienzo.create_rectangle(xactual,200,xactual+10,200-yactual,ou
tline="#ff0000",fill="#00ff00")
    xanterior = xactual
    yanterior = yactual


lienzo.pack(side=TOP)
```


tk

13. Uso de Librerías

13.1. Introducción a las librerías

13.2. Creación de nuestras propias librerías

Una de las principales ventajas de Python consiste en que podemos crear nuestras propias librerías. Para ello, cualquier archivo de Python, que hayamos programado previamente se puede convertir en una librería, simplemente llamándolo desde un archivo de origen.

Así pues, en este ejemplo, vamos a partir de un archivo que tendrá el siguiente contenido:

```
def miFuncion():
    print("esta es una función")
```

Este archivo anterior lo hemos guardado como “origen.py”.

Y a continuación, y en la misma carpeta, tendremos un segundo archivo de código, que realiza una llamada al primer archivo, de la siguiente forma.

```
from origen import *

miFuncion()
```

Si ejecutamos el código anterior, podremos comprobar en la consola cómo sale el resultado correcto de ejecutar la función personalizada dentro del archivo de destino.

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============= RESTART:
/Users/josevicente/Desktop/Python/destino.py ============
```

```
esta es una función
```

En este ejemplo, en el primer archivo, hemos declarado una función, para ser utilizada como librería por el segundo archivo, y ser llamada en ese momento.

Con Python ocurre que cuando utilizamos librerías, todas aquellas variables, todos aquellos métodos, y en definitiva, todo aquello que existe en la librería, automáticamente existe en el archivo de destino, pero debemos tener cuidado porque no ocurre así al revés.

13.3. Instalación de librerías

Python viene con una cantidad increíblemente grande de librerías pre instaladas.

Pero, además de esto, podemos encontrar en Internet, y podemos descargar fácilmente, una cantidad todavía mucho más grande de las librerías que vienen preinstaladas con la instalación estándar del entorno de desarrollo de Python.

Junto con el intérprete, se instala una herramienta que muchas veces es desconocida para el usuario novel, que se llama PIP.

Esta herramienta, a la cual se accede mediante la línea de comandos, o el terminal, nos permite realizar instalaciones de una forma sencilla.

Por ejemplo, en el caso de MySQL, para poder instalar la librería correspondiente, simplemente tenemos que abrir un nuevo terminal, y tenemos que introducir el siguiente comando.

```
pip install mysql-connector-python
```

Para poder conocer la lista completa de librerías disponibles, así como los comandos necesarios para instalarlas, podemos consultarlo en la web:

https://pypi.org/

13.4. Librería matemática

Dentro de Python encontramos, como en prácticamente cualquier lenguaje de programación moderno, una librería con una serie de funciones y constantes matemáticas que resultan imprescindibles para poder desarrollar correctamente una inmensa cantidad de programas.

Al igual que en otros lenguajes de la la librería matemática no se encuentra dentro del núcleo del propio lenguaje, más allá de los operadores aritméticos que hemos visto al con anterioridad, y, por tanto, mediante esta librería, podemos invocar a funciones que nos permiten, entre otras cosas:

- Calcular funciones trigonométricas,
- Calcular funciones de redondeo
- Obtener máximos y mínimos
- Obtener constantes Tales como por ejemplo el número pi y el número e
- Y otras muchas más.

El uso de la librería matemática es tremendamente sencillo, únicamente debemos importar la librería correspondiente, y a continuación podemos directamente llamar a cada uno de sus métodos o a las constantes que contiene.

Podemos encontrar la documentación completa acerca de las funciones. Y las constantes que contiene esta librería en la siguiente dirección que corresponde a la documentación oficial.

```
##Documentación sobre la librería matemática
##https://docs.python.org/3/library/math.html

import math

print(math.sqrt(9))

print(math.ceil(3.3))
```

```
print(math.sin(math.pi/2))
```

Al ejecutar el código anterior en la consola, podremos comprobar como el programa representa correctamente cada uno de los valores calculados utilizando las funciones matemáticas de raíz cuadrada, redondeo al alza, y cálculo del seno trigonométrico.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
/Users/josevicente/Desktop/Python/matematica.py ==========
3.0
4
1.0
```

13.5. Librería de tiempo

Otra de las operaciones, tremendamente constantes que encontramos a lo largo de diferentes programas informáticos en múltiples lenguajes de programación, es la capacidad de poder conocer y gestionar el tiempo.

La librería del tiempo nos permite, por ejemplo, conocer el tiempo actual, formatearlo, en cuanto a años, meses, días, horas, minutos, segundos, y también, por ejemplo, nos permite realizar otras funciones, Tales como por ejemplo detener el flujo de la ejecución de un programa durante un tiempo determinado, invocando a la función Sleep

```
import time

contador = 0
```

```
numero = 0.00000000004234


def bucle(contador,numero):
    contador += 1
    numero = numero*1.2
    time.sleep(0.0001)
    bucle(contador,numero)

time.sleep(1)
bucle(contador,numero)
```

Al final del ejercicio anterior hemos podido comprobar cómo creamos una función de bucle, que básicamente es una función que se llama a sí misma de forma recursiva.

Si ejecutamos esta función sin utilizar la librería de tiempo, nos encontraríamos que habríamos entrado en un bucle sin ningún tipo de control.

Sin embargo, utilizando una librería de tiempo, podemos decirle al programa que detenga la ejecución durante una cantidad concreta de tiempo, de tal forma que podemos controlar mucho mejor cómo se ejecutan los recursos del sistema utilizando este tipo de programación.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============= RESTART:
/Users/josevicente/Desktop/Python/tiempo1.py ============
```

13.6. Threading

Por defecto, todos los programas que escribimos, si no decimos lo contrario, tienen un modelo de ejecución bloqueante, en el cual un bloque de código no se ejecuta hasta que el bloque anterior ha finalizado su ejecución.

Este modelo de desarrollo muchas veces nos es altamente conveniente, ya que nos permite controlar el orden en el cual se ejecutan los bloques del código que contiene nuestro programa.

Sin embargo, hay ocasiones en las cuales nos interesa que haya bloques de código que se ejecuten, pero que el hilo de ejecución no quede bloqueado.

Byron nos propone múltiples soluciones para esta situación, donde una de ellas consiste en utilizar la librería de Frederic, que convierte Los bloques de código en hilos de ejecución, que se ejecutan de forma independiente y no bloquean la ejecución del programa principal.

En este ejemplo de código, en primer lugar, importamos la librería de Frederic, importamos la librería de tiempo y la librería matemática. La librería de Frank nos va a servir para crear hilos de ejecución, la librería de tiempo nos va a servir para detener la ejecución de cada uno de los hilos de una forma controlada, y la librería matemática nos va a servir para llamar a la constante PI.

A partir de aquí, creamos una función en la cual le pasamos un número, y dentro de la función observamos cómo se produce una operación matemática de una forma completamente artificial con el único objetivo de mantener ocupado al hilo de ejecución realizando un cálculo.

Podremos observar que en la última línea de la función, la función se llama asimismo de forma recursiva, entrando en un bucle infinito. Para evitar que ese bucle infinito bloquee el hilo de ejecución, hemos introducido una pequeña parada en la ejecución utilizando la función slip de la librería de tiempo.

A partir de ahí, en la siguiente parte del código, creamos una serie de hilos de ejecución, donde el número de hilos lo podemos encontrar dentro del

parámetro del rango del foro. A partir de ahí, creamos tantas tareas como sea necesario, y las arrancamos.

La idea de este código es que hemos creado una gran cantidad de tareas, que se ejecutan de forma recursiva, con el único objetivo de poder comprobar que dichas tareas están en ejecución, comprobando los niveles de carga del procesador en cualquiera de los sistemas operativos en los que estemos utilizando este programa.

```
import threading
import time
import math

def trabajador(numero):
    minumero = math.pi/(numero+1)
    time.sleep(0.1)
    trabajador(numero)

tareas = []

for i in range(2400):
    t = threading.Thread(target = trabajador,args=(i,))
    tareas.append(t)
    t.start()
```

Al ejecutar este programa, comprobaremos cómo arrancamos una serie de procesos de forma paralela, y cada uno de ellos realiza el cálculo que hemos propuesto en el ejercicio anterior, pero no de forma secuencial, sino repartiendo los procesos en los hilos de ejecución que están disponibles dentro del hardware de nuestro sistema informático.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

========== RESTART:
```

```
/Users/josevicente/Desktop/Python/threading.py ===========
```

13.7. Gráficas

Una de las librerías que más importancia ha cobrado en los últimos años, especialmente con la popularización de las técnicas de Big Data, es la librería *matplotlib*, mediante la cual podemos generar gráficas de una forma terriblemente sencilla, y a la vez potente, utilizando Python.

Python ha resultado ser un lenguaje de referencia en el terreno del Big Data y la inteligencia artificial, por su tremenda sencillez, comparativamente a otros lenguajes de programación, lo cual facilita el acceso a través de Python a estas disciplinas no tienen un pasado relacionado de forma estricta con la informática.

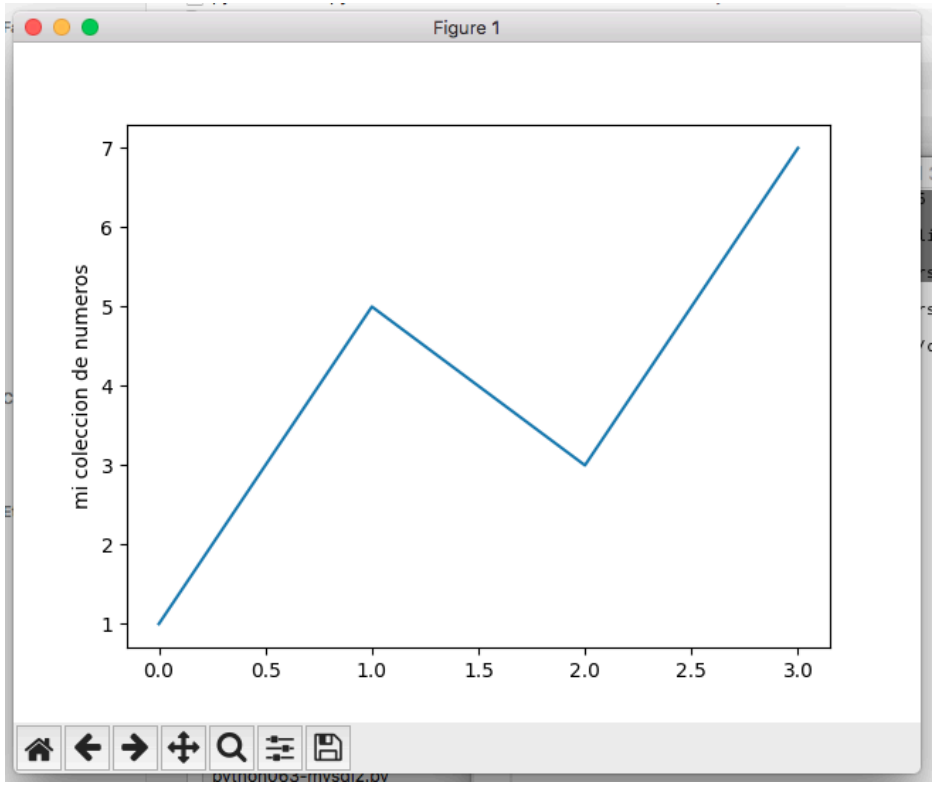
Incluso, tal vez, tú mismo o tú misma que estás leyendo esta obra, te podrás identificar con este perfil de usuario.

De esta forma, siendo Python un lenguaje de programación, terriblemente fácil de usar y de comprender, en el terreno de la visualización de la información, la librería *matplotlib*, en combinación con Python, proporciona a los usuarios una herramienta tremendamente fácil y poderosa para generar gráficas que nos permitan representar los datos de una forma sencilla y ligera.

A continuación, voy a presentar una serie de ejercicios en los cuales el objetivo es demostrar precisamente esta potencia y esta sencillez de uso.

```
import matplotlib.pyplot as dibujar
dibujar.plot([1,5,3,7])
dibujar.ylabel("mi coleccion de numeros")
dibujar.show()
```

Al ejecutar esta aplicación o tendremos una primera comprobación del resultado de trabajar con la librería de impresión de gráficas, y es que, más allá del resultado que salga en la consola, aparecerá una ventana flotante con la información que hemos introducido dentro de la gráfica de tipo lineal



13.7.1. Ejercicio: gráficas a partir de una base de datos

Generalmente, las gráficas representan conjuntos de datos, y los conjuntos de datos, en una gran cantidad de ocasiones, se encuentran almacenados en las bases de datos.

Es por esto que un ejercicio muy común en la generación de gráficas consiste en poder conectar la librería de creación de diferentes tipos de gráficas, con

una base de datos, para que el origen de datos de las gráficas sea aquellos datos contenidos dentro de la base de datos.

A continuación, y para ilustrar este caso tan frecuente, vamos a realizar un ejercicio mediante el cual, en primer lugar nos conectamos a una base de datos, extraemos información, y la representamos en pantalla en forma visual

```
import mysql.connector as my
import matplotlib.pyplot as plt

## Parte de la base de datos
mibd = my.connect(
    host = "localhost",
    port = "3306",
    user = "josevicente",
    password = "josevicente",
    database = "cursopython"
    )

micursor = mibd.cursor()

micursor.execute("SELECT COUNT(Nombre) AS cuenta, Nombre FROM
Alumnos GROUP BY Nombre ORDER BY cuenta DESC LIMIT 25")
miresultado = micursor.fetchall()



sizes = [0]

cadena = "'hola'"
for i in miresultado:

        sizes.append(i[0])
        cadena += ",'"+str(i[1])+"'"

labels = eval(cadena)
```

```
print("vamos a comprobar")
print(labels)
print(sizes)
print("quiero ver el tipo de dato")
print(type(labels))


fig1, ax1 = plt.subplots()
ax1.pie(sizes, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is
drawn as a circle.

plt.show()
```

Al ejecutar este ejemplo, obtendremos en pantalla la gráfica correspondiente a la petición que hemos solicitado utilizando la sentencia SQL. por lo tanto, la salida no será necesariamente mostrada en la consola, sino que, sobre todo, aparecerá una gráfica flotando como una ventana en nuestra interfaz de usuario

13.8. Numpy

Nupi es una librería muy conocida y muy utilizada dentro del mundo Python, ya que nos permite disponer de herramientas adicionales, estructuras de datos adicionales, y una colección de funciones matemáticas que nos permiten ampliar en gran medida el conjunto de herramientas que Python tiene de base.

En más de una ocasión dentro de este libro. Hemos comentado que la gran ventaja que tiene Python sobre otros lenguajes de programación es su facilidad y su ex sensibilidad, pero una gran desventaja, al menos a nivel Comparativo con respecto a otros lenguajes, es el rendimiento cuando trabajamos con grandes conjuntos de datos.

Una uno de los beneficios que nos ofrecen un pie cuando lo utilizamos con Python es precisamente tener la capacidad de seguir utilizando la facilidad de uso que nos aporta este lenguaje de programación, pero a la vez, nos permite utilizar estructuras de control que, cuando ejecutamos nuestros programas, nos proporcionan un rendimiento bastante mayor el que proporciona Python por defecto.

Por esto, a continuación vamos a presentar una serie de ejemplos, en los cuales analizaremos unas pocas funcionalidades de las muchas que nos ofrece esta librería.

13.8.1. Comprobación inicial

En primer lugar, vamos a crear una colección, que corresponde al concepto de lista en Python, pero para hacerlo vamos a llamar específicamente a una colección nativa de Numpy.

Para ello en primer lugar importamos la librería para comprobar que la tenemos disponible en el sistema.

```
import numpy as np
```

Si todo ha ido correctamente, en la consola obtendremos un resultado vacío, indicando que la librería ha sido importada exitosamente

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============== RESTART:
/Users/josevicente/Desktop/Python/numpy.py =============
```

A continuación vamos a crear un conjunto, y para ello lo haremos de la siguiente forma.

```
import numpy as np

coleccion =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

print(coleccion[0])
```

Al ejecutar este código en la consola, podemos comprobar como, en cuanto a la salida, las colecciones de la librería numpy se comportan de forma aparentemente igual que las listas en Python.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============= RESTART:
/Users/josevicente/Desktop/Python/numpy2.py =============

============= RESTART:
/Users/josevicente/Desktop/Python/numpy2.py =============
Juan
```

Por último queremos ser capaces de acceder al conjunto, y de imprimir el tipo de dato que está contenido dentro de esa colección, para lo cual utilizaremos los siguientes dos comandos.

```
import numpy as np
```

```
coleccion =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

print(coleccion[0])
print(type(coleccion))
```

Sin embargo, en esta siguiente ejecución podremos comprobar que , aunque el resultado en pantalla sea muy similar a trabajar con listas en el núcleo de Python, el sistema reconoce que el tipo de datos es un tipo específico de la librería numpy

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============= RESTART:
/Users/josevicente/Desktop/Python/numpy3.py =============

============= RESTART:
/Users/josevicente/Desktop/Python/numpy3.py =============
Juan
<class 'numpy.ndarray'>
```

13.8.2. Recortar con numpy

La librería Nupi nos ofrece múltiples herramientas para poder trabajar con cantidades de datos, así que en este ejercicio vamos a demostrar una de estas herramientas consistente en la capacidad de cortar una colección de objetos, pudiendo indicar el punto inicial de corte y el punto final.

Para esto, en este siguiente ejercicio en primer lugar volvemos a importar librería.

```
import numpy as np
```

A continuación creamos una colección de datos como colección nativa de numpy.

```
import numpy as np

coleccion =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])
```

Y en primer lugar la imprimimos para comprobar que tenemos todos los elementos de la colección.

```
import numpy as np

coleccion =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

print(coleccion)
```

A continuación simplemente utilizamos la siguiente instrucción para especificar que queremos cortar desde el primer elemento hasta el cuarto, teniendo en cuenta que los conjuntos de datos empiezan en el índice número cero.

```
import numpy as np

coleccion =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])
```

```
print(coleccion)

cortado = coleccion[0:3]
```

A continuación imprimimos el resultado en pantalla, para comprobar que hemos recortado el conjunto de datos inicial al introducirlo dentro de la nueva variable

```
import numpy as np

coleccion =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

print(coleccion)

cortado = coleccion[0:3]
print(cortado)

print(coleccion[2:5])
```

Si la operación se ha realizado de forma correcta, podremos comprobar como hemos podido recortar el conjunto de datos inicial en un nuevo subconjunto

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============= RESTART:
/Users/josevicente/Desktop/Python/numpy4.py =============

============= RESTART:
```

```
/Users/josevicente/Desktop/Python/numpy4.py =============
['Juan' 'Jorge' 'Jose' 'Julia' 'Javier' 'Jacobo']
['Juan' 'Jorge' 'Jose']
['Jose' 'Julia' 'Javier']
```

13.8.3. Concatenaciones

Otra de las instrucciones útiles y sencillas que nos propone la librería Nupi consiste en ser capaces de concatenar conjuntos diferentes de información.

Para realizar este ejercicio, en primer lugar, importamos la librería

```
import numpy as np
```

Y a continuación creamos dos conjuntos diferentes de datos, introducidos en dos identificadores llamados colección uno y colección dos

```
import numpy as np

coleccion1 =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

coleccion2 =
np.array(['Pablo','Pedro','Paco','Pio','Paloma'])
```

A continuación concatenamos las dos colecciones simplemente llamando a la siguiente instrucción

```
import numpy as np

coleccion1 =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

coleccion2 =
np.array(['Pablo','Pedro','Paco','Pio','Paloma'])


juntado = np.concatenate((coleccion1,coleccion2))
```

Y por último realizamos una Impresión por pantalla, simplemente para comprobar que la concatenación ha finalizado correctamente

```
import numpy as np

coleccion1 =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

coleccion2 =
np.array(['Pablo','Pedro','Paco','Pio','Paloma'])


juntado = np.concatenate((coleccion1,coleccion2))

print(juntado)
```

Comprobamos que la concatenación se ha realizado correctamente, mostrando el resultado a través de la terminal utilizando el último comando de impresión por pantalla, print

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
```

```
information.

============= RESTART:
/Users/josevicente/Desktop/Python/numpy5.py =============

============= RESTART:
/Users/josevicente/Desktop/Python/numpy5.py =============
['Juan' 'Jorge' 'Jose' 'Julia' 'Javier' 'Jacobo' 'Pablo'
'Pedro' 'Paco'
 'Pio' 'Paloma']
```

13.8.4. Partir estructuras con numpy

En este siguiente ejercicio, vamos a probar uno de los métodos que nos proporciona numpy para trabajar con la partición de grandes bloques de datos.

```
import numpy as np

coleccion1 =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

coleccion2 =
np.array(['Pablo','Pedro','Paco','Pio','Paloma'])


juntado = np.concatenate((coleccion1,coleccion2))

print(juntado)
```

Partiendo desde el código anterior, y partiendo desde la concatenación resultante del ejercicio anterior, a continuación vamos a crear un nuevo identificador, llamado separado, y pondremos la siguiente instrucción para tomar la lista de elementos y partirla en tres partes lo más iguales posibles.

```
import numpy as np

coleccion1 =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

coleccion2 =
np.array(['Pablo','Pedro','Paco','Pio','Paloma'])


juntado = np.concatenate((coleccion1,coleccion2))

print(juntado)

separado = np.array_split(juntado,3)
```

Por último, vamos a introducir los siguientes comandos de impresión para comprobar cuál es la partición que ha hecho la instrucción con respecto a los datos originales.

```
import numpy as np

coleccion1 =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

coleccion2 =
np.array(['Pablo','Pedro','Paco','Pio','Paloma'])


juntado = np.concatenate((coleccion1,coleccion2))

print(juntado)

separado = np.array_split(juntado,3)
```

```
print("Que sepas que la primera parte del partido es")
print(separado[0])
print("Que sepas que la segunda parte del partido es")
print(separado[1])
print("Que sepas que la tercera parte del partido es")
print(separado[2])
```

Veremos que el programa ha hecho todo lo posible para tomar un conjunto de datos y partirlo en tantas partes lo más iguales posibles, como le haya sido posible, lo cual tiene grandes beneficios, especialmente en los programas que trabajan con múltiples núcleos, y tienen que partir grandes cantidades de información.

```
import numpy as np

coleccion1 =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

coleccion2 =
np.array(['Pablo','Pedro','Paco','Pio','Paloma'])


juntado = np.concatenate((coleccion1,coleccion2))

print(juntado)

separado = np.array_split(juntado,3)
print("Que sepas que la primera parte del partido es")
print(separado[0])
print("Que sepas que la segunda parte del partido es")
print(separado[1])
print("Que sepas que la tercera parte del partido es")
print(separado[2])
```

Al ejecutar esta parte del código podremos comprobar como la función de Split parte la colección inicial de elementos en las partes más equitativas posibles

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

===== RESTART:
/Users/josevicente/Desktop/cursodepython/coleccionesnumpy.py
====

===== RESTART:
/Users/josevicente/Desktop/cursodepython/coleccionesnumpy.py
====
['Juan' 'Jorge' 'Jose' 'Julia' 'Javier' 'Jacobo' 'Pablo'
'Pedro' 'Paco'
 'Pio' 'Paloma']
Que sepas que la primera parte del partido es
['Juan' 'Jorge' 'Jose' 'Julia']
Que sepas que la segunda parte del partido es
['Javier' 'Jacobo' 'Pablo' 'Pedro']
Que sepas que la tercera parte del partido es
['Paco' 'Pio' 'Paloma']
```

13.8.5. Búsquedas con numpy

Otra de las funciones tremendamente útiles que podemos encontrar dentro de esta librería, nos permite realizar búsquedas dentro de las estructuras de datos.

Para ello, partiendo de nuevo del resultado del ejercicio anterior, vamos a crear una colección como unión de dos colecciones anteriores, y a continuación utilizaremos el siguiente comando para buscar un nombre dentro del listado de nombres Con el que estamos trabajando dentro de estos ejercicios.

A continuación introduciremos un comando print, para imprimir por pantalla, el resto de la petición, comprobando que el sistema, de esta forma, es capaz de localizar el índice del elemento que vamos buscando

```
import numpy as np

coleccion1 =
np.array(['Juan','Jorge','Jose','Julia','Javier','Jacobo'])

coleccion2 =
np.array(['Pablo','Pedro','Paco','Pio','Paloma'])


juntado = np.concatenate((coleccion1,coleccion2))

busqueda = np.where(juntado == 'Javier')
print(busqueda)
```

Al ejecutar el ejemplo anterior, podremos comprobar que , en primer lugar, realizamos una concatenación, y a continuación una búsqueda dentro de los resultados de la concatenación anterior.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

============= RESTART:
/Users/josevicente/Desktop/Python/numpy6.py =============

============= RESTART:
/Users/josevicente/Desktop/Python/numpy6.py =============
(array([4]),)
```

13.9. Expresiones regulares

13.9.1. Expresiones regulares

Las expresiones regulares son un conjunto de herramientas que nos permiten validar una cadena de caracteres, con criterios que van mucho más allá de una simple comprobación de igualdad, o de comprobar si una parte de una cadena se encuentra dentro de otra cadena determinada.

Nos permiten realizar validaciones en base a patrones, y para ello, existe un estándar en cuanto a la forma en la que podemos realizar dicha validación de las expresiones, para conseguir diferentes resultados.

A continuación te presentamos una serie de ejemplos para aprender el fundamento de cómo trabajar con expresiones regulares.

En primer lugar, deberemos tener en cuenta que para utilizar las expresiones regulares debemos importar la librería R e dentro de Python.

Una vez que hemos importado la librería, a continuación definimos dos variables, la primera de ellas, siendo el texto dentro del cual queremos realizar una búsqueda, y la segunda de ellas la que contiene el resultado de la verificación de la más sencilla de las dos personas regulares que vamos a mostrar en los ejercicios de esta publicación, en la que buscamos un trozo de una cadena dentro de otra cadena.

```
import re

mitexto = "Segismundo"
busqueda = re.search("^Se",mitexto)
```

El resultado de la variable llamada, en este caso, búsqueda, al fin, y al cabo, es un resultado muy lejano acerca de sí, se ha conseguido encontrar la cadena dentro de la cadena superior, y, por lo tanto, si lo imprimimos en

pantalla, observaremos cómo obtenemos un resultado basado en *true* o *false*.

```
import re

mitexto = "Segismundo"
busqueda = re.search("^Se",mitexto)
print(busqueda)
```

Al ejecutar este programa , obtendremos como resultado en pantalla la validación, verdadera o falsa, acerca de si se ha conseguido encontrar el texto que se estaba buscando, dentro de la cadena de referencia.

No solo eso, sino que además podemos introducir esa variable dentro de una estructura de control condicional de tipo F, para capturar cada uno de los casos verdadero o falso, y ejecutar un bloque de códigos, según el resultado que nos ha llegado la expresión anterior

```
import re

mitexto = "Segismundo"
busqueda = re.search("^Se",mitexto)
print(busqueda)
if busqueda:
    print("he encontrado un resultado")
else:
    print("no he encontrado ningún resultado")
```

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.
```

```
===== RESTART:
/Users/josevicente/Desktop/cursodepython/python080-regexp.py
====
<re.Match object; span=(0, 2), match='Se'>
he encontrado un resultado
```

13.9.2. Validar campo

Una función muy común de las expresiones regulares consiste en poder validar, y en un momento dado, incluso poder sanear, toda la información que proviene de las entradas de los usuarios.

De esta forma, en este siguiente ejercicio lo que haremos es comprobar si la información que ha introducido un usuario es un número, por ejemplo, en el punto de entrada a una aplicación, ya que si creamos una aplicación que espera encontrarse un número, y el usuario introduce por ejemplo una letra o un carácter, nuestro programa podía fallar.

Para ello, en primer lugar, creamos un programa en el cual se le pide al usuario final que introduzca un número, Que se almacena en una variable, y a continuación, especificamos una regla

```
import re

print("Introduce un numero")
email = input()
regla = r'^([\s\d]+)$'
```

Una vez que el usuario ha realizado dicha entrada, podemos crear una nueva variable llamada validación, en la cual realizamos una búsqueda del patrón, dentro de la regla que hemos generado previamente.

```
import re
```

```
print("Introduce un numero")
email = input()
regla = r'^([\s\d]+)$'

validacion = re.search(regla,email)
```

Al ejecutar este ejemplo de código, podremos comprobar como, gracias a la regla de validación mediante expresiones regulares, podemos obtener en pantalla si aquello que hemos introducido dentro de la función input es realmente un número, o no lo es

Y por último, podemos crear una estructura de control condicional Iv para conseguir controlar si la información que ha introducido el usuario es correcta con respecto a nuestro objetivo, y a partir de ahí continuar la ejecución del programa de una forma o de otra.

```
import re

print("Introduce un numero")
email = input()
regla = r'^([\s\d]+)$'

validacion = re.search(regla,email)

if validacion:
    print("ok")
else:
    print("no ok")
```

Comprobaremos en la consola como no solo podemos atrapar el valor en formato verdadero o falso con respecto a si lo que hemos introducido en la entrada es un número o no lo es, sino que podemos introducir esa información dentro de una estructura de control, por ejemplo una estructura de control condicional, y a partir de ahí elegir el código que ejecutamos

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

== RESTART:
/Users/josevicente/Desktop/cursodepython/python081-validarca
mpo.py =
Introduce un numero
3
ok
```

13.9.3. Validar email

Otra de las funciones tremendamente comunes de las expresiones regulares consiste en la validación de campos complejos, tales como por ejemplo los campos de correo electrónico.

En cualquier aplicación informática deben estar correctamente introducidos. Los correos electrónicos de los usuarios, puede ser muy importante desde el punto de vista en el que si el correo electrónico del usuario no está correctamente introducido, más adelante, podemos tener problemas para poder comunicarnos con los usuarios.

En este siguiente ejemplo mostramos una validación de correo electrónico, y para ello, en primer lugar, importamos la librería que nos permite utilizar, expresiones regulares, y a continuación permitimos que el usuario introduzca un correo electrónico, tras lo cual creamos una regla, vas a ver una expresión regular para validar, de forma genérica, correos electrónicos.

```
import re

print("Dime un correo electronico")
email = input()
```

```
regla = '^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,3}$'

validacion = re.search(regla,email)
```

A continuación realizamos una validación de la variable que hemos almacenado previamente, para capturar, si ha sido realizada de forma correcta, y que el programa realice una serie de operaciones en el caso verdadero, y otra serie de operaciones en el caso falso.

```
import re

print("Dime un correo electronico")
email = input()
regla = '^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,3}$'

validacion = re.search(regla,email)

if validacion:
    print("Lo que has introducido es un correo electronico y
lo voy a meter en la base de datos")
else:
    print("Lo que has introducido no es un correo
electronico")
```

Al ejecutar este código, el sistema preguntará al usuario que introduzca una dirección de correo electrónico, y a continuación realizará la validación.

Deberemos tener en cuenta que las reglas de validación de correos electrónicos mediante expresiones regulares son uno de los grandes retos a lo largo de los años, no existiendo una única e infalible regla para poder validar todos los correos electrónicos del mundo.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
```

```
Type "help", "copyright", "credits" or "license()" for more
information.

== RESTART:
/Users/josevicente/Desktop/cursodepython/python081-validarem
ail.py =
Dime un correo electronico
info@josevicentecarratala.com
Lo que has introducido es un correo electronico y lo voy a
meter en la base de datos
```

13.10. JSON

El formato Json, como formato de intercambio de datos, está completamente extendido en la actualidad, y por tanto es muy importante que sepamos cómo tratar información que provenga en este formato, cuando creamos programas en el lenguaje de programación Python, ya que es muy probable que nuestros programas, tarde o temprano, tenga la necesidad de tratar información guardada en este formato

13.10.1. JSON

En este primer ejemplo empezamos partiendo de una cadena que contiene una serie de piezas de información correctamente formateada dentro de la cadena de caracteres alfanuméricos.

En primer lugar también deberemos darnos cuenta de que para poder trabajar con Json, deberemos, de forma previa, haber importado la librería correspondiente dentro de nuestro programa.

```
import json


micadena =
'{"Juan":"juan@correo.com","Jorge":"jorge@correo.com","Javie
```

```
r":"javier@correo.com","Julia":"julia@correo.com","Jacobo":"
jacobo@correo.com"}'
```

A continuación utilizamos la instrucción de carga, que nos permite convertir la cadena a un elemento de tipo Json.

```
import json


micadena =
'{"Juan":"juan@correo.com","Jorge":"jorge@correo.com","Javie
r":"javier@correo.com","Julia":"julia@correo.com","Jacobo":"
jacobo@correo.com"}'

carga = json.loads(micadena)
```

Para continuar, vamos a imprimir tanto el tipo de datos que contiene la cadena, como el tipo de datos, después de haberlo convertido en una estructura de datos de tipo Json, para encontrar, con sorpresa, que el sistema convierte de forma muy fácil una cadena alfanumérica con formato Json, en un diccionario de Python.

```
import json


micadena =
'{"Juan":"juan@correo.com","Jorge":"jorge@correo.com","Javie
r":"javier@correo.com","Julia":"julia@correo.com","Jacobo":"
jacobo@correo.com"}'

carga = json.loads(micadena)
print(type(micadena))
print(type(carga))
```

Por lo tanto, una vez que hemos realizado esta conversión, es cuando podemos llamar a cualquier elemento del archivo Json de la misma forma que lo haríamos llamando a un elemento del diccionario, simplemente realizando una llamada, introduciendo el índice alfanumérico del diccionario al que queremos acceder

```
import json


micadena =
'{"Juan":"juan@correo.com","Jorge":"jorge@correo.com","Javie
r":"javier@correo.com","Julia":"julia@correo.com","Jacobo":"
jacobo@correo.com"}'

carga = json.loads(micadena)
print(type(micadena))
print(type(carga))

print(carga["Juan"])
```

Podremos comprobar que mientras que el tipo de la cadena es una cadena alfanumérica, el tipo de la carga es un archivo JSON.

Una vez que el archivo ha sido cargado, podemos acceder a cualquiera de los elementos de la misma forma que vemos cuando queremos tratar con diccionarios. realizamos una llamada al identificador carga, y le pasamos el índice asociativo dentro de ". El resultado en la consola será el siguiente.

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.

====== RESTART:
```

```
/Users/josevicente/Desktop/cursodepython/python082-json.py
=====
<class 'str'>
<class 'dict'>
juan@correo.com
```

13.10.2. Leer un JSON externo

En el ejercicio anterior, hemos creado una cadena con formato Json de forma completamente interna a nuestro programa.

Sin embargo, esta situación, que hemos creado de forma artificial para demostrar de la forma más sencilla el trabajo con este tipo de información, realmente no representa, o no suele representar, la situación que nos vamos a encontrar de forma más común en nuestro día a día.

Generalmente los archivos jotas son si sirven como archivos externos a nuestro programa, que debemos cargar de forma dinámica. Y ello lo podemos conseguir utilizando las instrucciones de carga de archivos que hemos visto previamente dentro de esta publicación.

Para ello, en este primer ejercicio, lo que hacemos es importar la librería correspondiente para poder interpretar archivos, Json, y a continuación, vamos a cargar, utilizando una estructura de datos, que en este caso llamaremos archivo, el contenido de un archivo llamado agenda. Json, el cual abriremos a modo de lectura.

```
import json

archivo = open("agenda.json",'r')
```

Una vez que hemos realizado esta operación, a continuación cargamos la única línea que contiene el archivo dentro de una cadena de caracteres.

```
import json

archivo = open("agenda.json",'r')

micadena = archivo.readline()
```

Y una vez que hemos conseguido este objetivo, es cuando ya, por fin, vamos a poder utilizar la instrucción aprendida en el ejercicio anterior, para convertir esa cadena de caracteres alfanuméricos en una estructura de datos de tipo Json,

```
import json

archivo = open("agenda.json",'r')

micadena = archivo.readline()

carga = json.loads(micadena)
```

Y a partir de ese momento ya podemos, en primer lugar, comprobar el formato de la estructura de datos que hemos cargado dos

```
import json

archivo = open("agenda.json",'r')

micadena = archivo.readline()

carga = json.loads(micadena)
print(type(micadena))
print(type(carga))
```

Por último, siendo capaces de recuperar un elemento de dicha cadena Json.

```
import json

archivo = open("agenda.json",'r')

micadena = archivo.readline()

carga = json.loads(micadena)
print(type(micadena))
print(type(carga))

print(carga["Jose"])
```

Al ejecutar el ejercicio, podremos comprobar como somos capaces no solo de obtener el tipo de datos de cada uno de los elementos, sino también de acceder a los índices asociativos del archivo JSON, accediendo a ellos como si se tratase de un diccionario nativo de Python.

```
{"Juan":"juan@correo.com","Jorge":"jorge@correo.com","Javier
":"javier@correo.com","Julia":"julia@correo.com","Jacobo":"j
acobo@correo.com","Jose":"jose@correo.com"}
```

```
Python 3.11.1 (v3.11.1:a7a450f84a, Dec  6 2022, 15:24:06)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more
information.
```

```
====== RESTART:
/Users/josevicente/Desktop/cursodepython/python082-json.py
=====
<class 'str'>
<class 'dict'>
juan@correo.com
```

13.10.3. Cargar un JSON multilínea

Si el ejercicio anterior ha intentado representar de la forma más realista posible, el proceso de carga de un archivo JUNJIN externo, el ejercicio que presentamos a continuación ilustra este proceso de una forma si cabe todavía más realista todavía.

En el ejemplo anterior hemos supuesto que el archivo Json recargamos de forma externa, contenía toda su información en una única línea.

Sin embargo, en la gran mayoría de casos los archivos Json que vamos a cargar tienen múltiples líneas.

Es por esto que en el ejemplo que presentamos a continuación, utilizamos una de las funciones de tratamiento de cadenas, en este caso, una función de reemplazado de datos, para conseguir convertir el formato de un archivo basado en múltiples líneas, en una variable que tenga únicamente una sola línea.

La primera parte del ejercicio, al igual que en el ejercicio anterior, en primer lugar, importamos la librería, y a continuación, cargamos una serie de datos contenidos en un archivo externo que abrimos en modo lectura.

```
import json

archivo = open("alumnos.json",'r')
```

A continuación cargamos el contenido de dicho archivo dentro de una variable, pero en este caso utilizamos también una instrucción Replays para cambiar los saltos de línea, representados por el carácter\N, por un espacio vacío, con lo que, de esta forma, conseguimos Tener toda la información de ese archivo en una sola línea alfanumérica.

```
import json

archivo = open("alumnos.json",'r')

micadena = archivo.read().replace('\n', '')
```

A partir de ahí es cuando podemos invocar al método de carga del archivo Json.

```
import json

archivo = open("alumnos.json",'r')

micadena = archivo.read().replace('\n', '')

carga = json.loads(micadena)
```

Podemos comprobar, como de costumbre, los tipos de datos tanto de la cadena alfanumérico de caracteres como del archivo que acabamos de cargar

```
import json

archivo = open("alumnos.json",'r')

micadena = archivo.read().replace('\n', '')

carga = json.loads(micadena)
```

```
print(type(micadena))
print(type(carga))
```

Y por último, comprobamos que podemos acceder a cualquiera de los elementos del archivo Json.

```
import json

archivo = open("alumnos.json",'r')

micadena = archivo.read().replace('\n', '')

carga = json.loads(micadena)
print(type(micadena))
print(type(carga))
print(carga[‘uno’]['dos']['tres'])
```

Una vez que ejecutamos estoy código vamos a poder comprobar cómo somos capaces de acceder a un archivo JSON externo, cargarlo dentro de una estructura de datos de Python, y a partir de ahí, acceder a cada uno de los índices asociativos de la estructura de datos, independientemente de la complejidad que tenga el archivo XML

13.11. Librería de sistema

Python incorpora una librería del sistema, que nos permite comunicarnos con el sistema operativo, huésped, en el cual se está ejecutando nuestra aplicación, y nos permite ejecutar diferentes tipos de comandos relacionados con dicho sistema operativo.

Llegados a este punto debemos tener cuidado con la ejecución de esta librería, ya que Las instrucciones a las que podemos llamar, o las operaciones que podemos realizar con el sistema operativo dependen en gran medida del tipo de sistema operativo con el que estemos trabajando, y por lo tanto, dentro de que Python tiene la naturaleza y la filosofía de ser

multiplataforma, Usando esta librería, podemos incurrir en códigos que sean específicos para un sistema operativo concreto, lo cual haría que nuestro programa no fuera compatible con otros sistemas operativos.

Si este es el caso, debemos tener en cuenta que la propia librería ver sistema operativo puede utilizarse para comprobar y controlar en qué sistema operativo se está ejecutando, y por lo tanto Nos puede servir para atrapar la información del sistema operativo y ejecutar unos bloques de código UOU Tros, en función del sistema con el cual estemos trabajando.

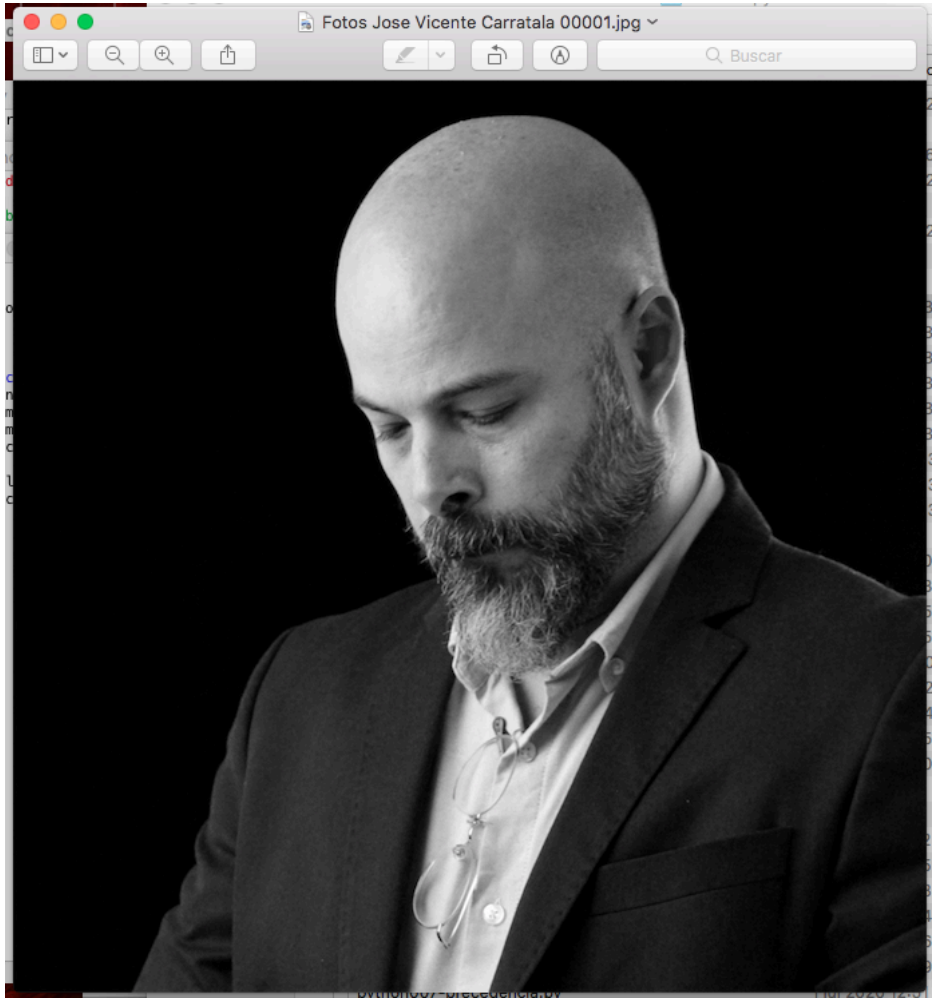
13.11.1. Recorrer

Una de las funciones muy útiles que tiene la librería del sistema consiste en utilizar la instrucción Work para ser capaces de entrar a un directorio y recorrer uno a uno los archivos contenidos, para realizar las operaciones que sean necesarias.

```
import os
from PIL import Image
import math
contador = 0
for root, dirs, files in os.walk("./fotos/retocadas/"):
    for filename in files:
        print(filename)
        if contador > 0:
                                                imagen     =
Image.open("./fotos/retocadas/"+filename)
            pixeles = imagen.load()
            altura = imagen.size[1]
            anchura = imagen.size[0]
            for x in range(0,anchura):
                for y in range(0,altura):
                    rojo = pixeles[x,y][0]
                    verde = pixeles[x,y][1]
                    azul = pixeles[x,y][2]
                    color = math.floor((rojo+verde+azul)/3)
```

```
                    rojo = color
                    verde = color
                    azul = color
                    pixeles[x,y] = (rojo,verde,azul)
            imagen.save("./fotos/retocadas/"+filename)
        contador += 1
```

Mediante la ejecución de este ejercicio podemos comprobar como, en primer lugar, somos capaces de acceder a una carpeta del sistema, y una vez que hemos accedido a esa carpeta, podemos recorrer uno a uno los archivos, en este caso para realizar una operación mediante el cual convertimos un conjunto de imágenes en color, a un conjunto ver imágenes en escala de grises.



13.12. PIL

La librería Pil, o lo que es lo mismo, la librería de procesamiento de imágenes de Python, es un conjunto de instrucciones que nos permiten tratar imágenes basadas en píxeles dentro de nuestros programas realizados con el lenguaje de programación Python.

El tratamiento de imágenes es una parte muy importante del desarrollo de programas informáticos, con lo cual constituye una pieza muy importante para permitirnos construir una gran cantidad de aplicaciones informáticas.

Lo primero que debemos tener en cuenta es que esta librería no suele venir preinstalada por defecto dentro de la distribución de Python, con lo cual deberemos instalarla abriendo un terminal e introduciendo el siguiente comando

Una vez que tenemos la librería instalada dentro del sistema, vamos a continuación a presentar una serie de ejemplos para demostrar su uso y para mostrar la gran potencia que nos ofrece esta librería.

13.12.1. Importar PIL

En primer lugar, y antes de empezar y realizar cualquier otra operación, es altamente recomendable que creemos un primer programa donde realizamos una comprobación inicial de si la librería ha sido correctamente importada.

Debemos tener en cuenta también que cada vez que instalamos nuevas librerías es altamente recomendable, aunque no es obligatorio, cerrar el intérprete de Python y volverlo a abrir de nuevo, Para asegurarnos de cargar correctamente todas las nuevas librerías que hayamos instalado dentro del sistema.

```
from tkinter import *
import PIL
```

13.12.2. Acceso a los píxeles

Una vez que hemos importado esta librería, vamos a realizar una serie de operaciones para conseguir demostrar su uso.

En primer lugar, una vez que hemos importado la librería, realizamos una importación de los comandos específicos para tratar imágenes, y cargamos una imagen que haya al lado de nuestro script para poder comprobar que tenemos acceso a toda su información.

Para este siguiente ejercicio partimos del supuesto de que tenemos una imagen llamada josevicente..jpg justo al lado del archivo Python en el que estemos trabajando en este momento.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
¡
```

Una vez que hemos conseguido ese objetivo, cargaremos la imagen dentro de una variable llamada píxeles, y para comprobar que la imagen se ha cargado correctamente, haremos un print en la consola del tamaño de la imagen, simplemente llamando a la propiedad *size*.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)
```

El objetivo principal de este ejercicio consiste en comprobar que podemos realizar dos operaciones muy importantes en el trabajo con píxeles, en primer lugar, saber que podemos leer el color de 1 px, y en segundo lugar, comprobar que podemos sobreescribir dicho valor.

Para ello, en la siguiente parte del ejercicio, queremos averiguar cuál es el color que podemos encontrar en el píxel 0,0, es decir, es el píxel que se

encuentra en la coordenada cero en el eje X, y en la coordenada cero en el eje Y.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])
```

Debemos tener en cuenta que las imágenes basadas en píxeles al final son rejillas bidimensionales de píxeles donde la primera dimensión es la X y la segunda dimensión es la Y.

También observaremos que al realizar esta petición en pantalla, dentro de la consola obtenemos como resultado una tupla.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])

pixeles[0,0] = (255,255,255)
```

En primer lugar es una tupla porque un píxel únicamente puede tener tres o cuatro componentes, según el formato que tenga la imagen, pero si hemos

escogido un formato JPG, la tupla que devuelve como resultado, es un conjunto de datos que representa el color rojo del píxel, el color verde del píxel, y el color azul del píxel.

En la siguiente parte del ejercicio comprobamos como no solo podemos obtener los índices de color de cada uno de los componentes de un píxel concreto, sino que los podemos igualar simplemente utilizando el operador de asignación.

Al igual que se ha comentado justo en la parte anterior de este ejercicio, los nuevos componentes que le asignó al píxel, deberán estar en formato de tupla.

De esta forma, podemos comprobar como lo que estoy haciendo es cambiar el color del primer píxel que se encuentra en la esquina superior izquierda de la imagen, es decir, el píxel que está en el valor de X igual a cero, y en el valor de Y igual a cero, y le estoy introduciendo el color blanco. El color blanco viene definido por un color tal que tiene el valor de 255 en el canal rojo, el valor 255 LMK al verde, y el valor 255 en el canal azul.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])
```

Por último en esta parte del ejercicio debemos guardar la imagen, ya que si no persistimos la información, hemos realizado el cambio en la memoria, pero no hay forma de que el usuario final pueda comprobar la efectividad del programa.

Es por esto que utilizamos la instrucción seis dentro de la librería, que nos permite guardar la Imagen en el nombre y con la ubicación que queramos.

En este caso lo que vamos a hacer es elegir un nombre diferente al nombre que tenía la imagen originalmente, para mantener la imagen original, y generar un duplicado, modificado.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])

pixeles[0,0] = (255,255,255)

imagen.save("josevicenteguardado.png")
```

Ejecutar este código no devolverá necesariamente un resultado en la consola punto por el contrario, deberemos abrir la imagen resultante en un programa de edición imágenes, deberemos realizar un zoom generoso en la esquina superior izquierda, y al hacerlo, comprobaremos como el primer pixel situado arriba a la izquierda en la imagen, habrá sido coloreado en blanco.

13.12.3. Recorrer píxeles - Imagen negativa

Una de las operaciones más comunes que suelen realizarse en programas que tratan imágenes, consiste en recorrer, uno a uno, los píxeles de dicha imagen, por ejemplo para realizar algún tipo de operación.

Cuando eso ocurre, teniendo en cuenta lo anteriormente mencionado, en cuanto a que las imágenes basadas en píxeles, realmente son matrices, bidimensionales de puntos, dispuestos en un eje horizontal al que llamamos X, y un eje vertical al que llamamos Y, una de las metodologías más comunes para resolver este barrido consiste en recorrer a un doble bucle Ford, que es Exactamente lo que vamos a hacer en el ejercicio que se presenta dentro de este apartado.

Tenemos en mente y en consideración que, dentro del llamado fin de Python, uno de los principios consiste en no anidar, dentro de lo posible, estructuras de control.

Aún teniendo en cuenta esa consideración, para este ejercicio, no queda más remedio que realizar dicha animación, y para ello vamos a proceder a desarrollar el ejercicio en su globalidad.

En primer lugar, como siempre, comenzamos importando la librería de tratamiento de imágenes, importando la su librería concreta para trabajar con imágenes.

```
from PIL import Image
```

A continuación cargamos una imagen y la introducimos dentro de una estructura de datos, para, a continuación, cargar todos sus píxeles, utilizando el método load.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)
```

Dado que vamos a estar utilizando un doble bucle en el cual recorremos la imagen primero en horizontal y luego en vertical, necesitamos conocer cuál es la anchura y la altura de dicha imagen, para lo cual tendremos en cuenta que la propiedad Saiz nos devuelve una tabla con dos valores, siendo el primero de ellos la anchura, y el segundo de ellos la altura.

Es por esto que crearemos dos variables de tipo número entero en los cuales introduciremos la información correspondiente a la anchura y la altura de la imagen.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])

altura = imagen.size[1]
anchura = imagen.size[0]
```

A continuación es cuando recorremos la imagen primero en horizontal, y luego, de forma animada, en vertical, para repasar, uno a uno, los píxeles de la imagen.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])

altura = imagen.size[1]
anchura = imagen.size[0]

for x in range(0,anchura):
    for y in range(0,altura):
```

Tengamos en cuenta que cada píxel viene representado por una tabla que contiene tres valores, siendo el primero de ellos el componente rojo de ese color, el segundo de ellos el componente verde, y el tercero de ellos el componente azul.

Estos componentes vienen representados, respectivamente, por los índices 0,1, y dos de la tupla que se encarga de atrapar el color.

Es por esto que en el siguiente bloque del código lo que hacemos es asociar cada uno de estos valores a una variable que internamente es un número entero.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])

altura = imagen.size[1]
anchura = imagen.size[0]

for x in range(0,anchura):
    for y in range(0,altura):
        rojo = pixeles[x,y][0]
        verde = pixeles[x,y][1]
        azul = pixeles[x,y][2]

        rojo = 255-rojo
        verde = 255-verde
        azul = 255-azul
```

```
        pixeles[x,y] = (rojo,verde,azul)
```

A continuación, para demostrar que tenemos control sobre cada uno de los píxeles de la imagen, he introducido una pequeña operación en la cual cogemos el valor rojo, verde, y azul de cada píxel de la imagen, y se lo restamos al valor 255, que es la máxima cantidad de información que tiene cada uno de los canales de la imagen.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])

altura = imagen.size[1]
anchura = imagen.size[0]

for x in range(0,anchura):
    for y in range(0,altura):
        rojo = pixeles[x,y][0]
        verde = pixeles[x,y][1]
        azul = pixeles[x,y][2]

        rojo = 255-rojo
        verde = 255-verde
        azul = 255-azul

        pixeles[x,y] = (rojo,verde,azul)
```

```
imagen.save("josevicenteguardado.png")
```

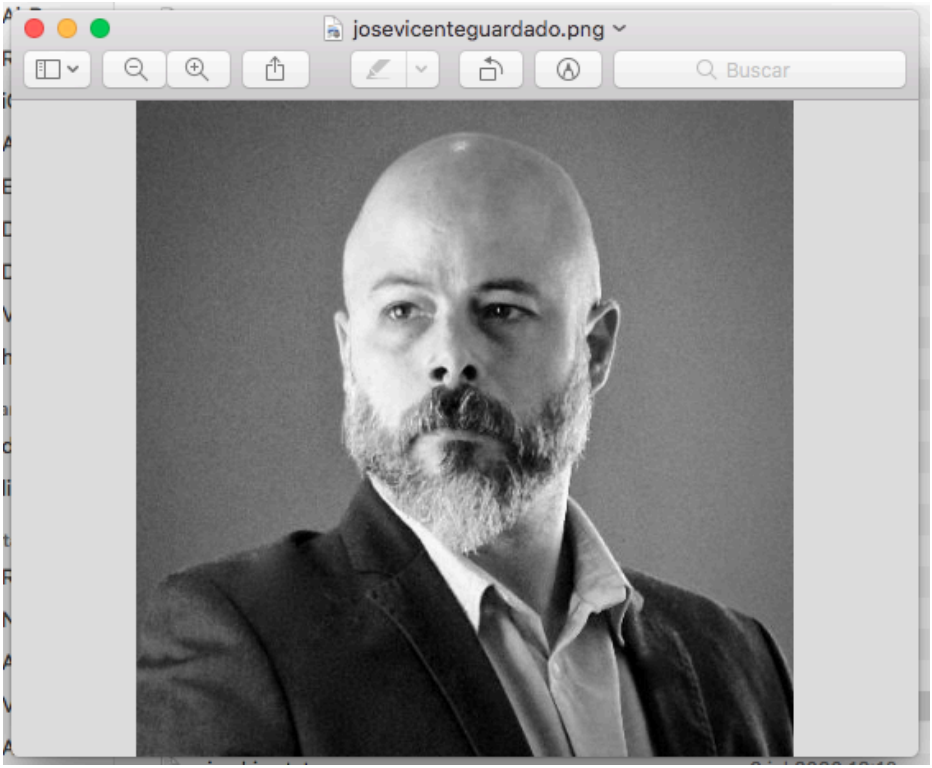
Con esto, lo que estamos consiguiendo es generar la versión negativa de color de esa imagen, como demostración de que estamos siendo capaces de acceder a los valores de color de cada uno de los píxeles.

Por último, utilizamos el comando Save para guardar la imagen correctamente dentro del disco, y si ejecutamos esta programación, podremos comprobar como el archivo resultante, corresponde a una versión negativa en cuanto al color con respecto a la imagen original.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])

altura = imagen.size[1]
anchura = imagen.size[0]

for x in range(0,anchura):
    for y in range(0,altura):
        rojo = pixeles[x,y][0]
        verde = pixeles[x,y][1]
        azul = pixeles[x,y][2]

        rojo = 255-rojo
        verde = 255-verde
        azul = 255-azul
```

```
        pixeles[x,y] = (rojo,verde,azul)


imagen.save("josevicenteguardado.png")
```

Si ejecutamos este código, podremos comprobar como en el terminal no obtendremos ningún resultado específico, pero sin embargo, si abrimos la imagen de destino, y el programa ha funcionado correctamente , obtendremos la misma imagen que tenemos en el origen, pero con los tres canales de color invertidos.



13.12.4. Recorrer píxeles grises - método 1

Otro ejercicio que podemos plantear en base al código que estamos desarrollando, consiste en crear una versión en escala de grises de la imagen que hemos cargado originalmente.

La estructura de este ejercicio es prácticamente igual que en el ejercicio anterior, donde la única diferencia consiste en que dentro del doble bucle Ford, podremos comprobar que en primer lugar tomamos Los tres componentes de color de 1 px, pero a continuación, al canal rojo, le asignamos el color rojo, pero al canal verde le asignamos igualmente el color rojo, y al canal azul le asignamos de nuevo el color rojo una vez más.

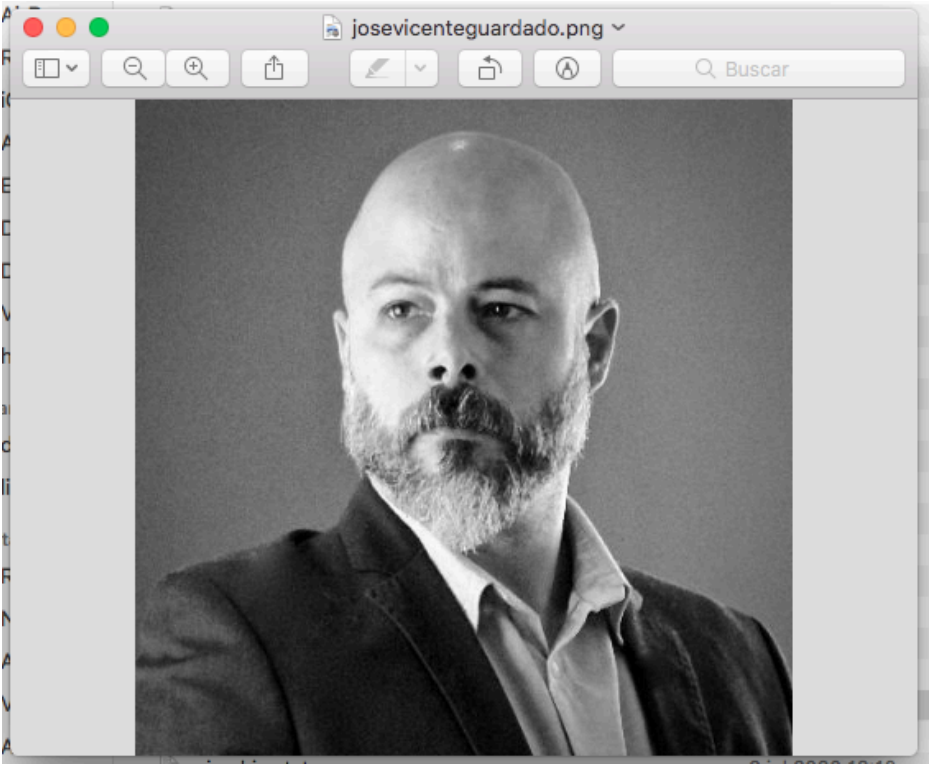
Por lo tanto, al final, lo que hacemos es vulgar el contenido del canal rojo, en los tres canales de color, y sobre escribir el color del píxel en el que nos encontramos actualmente.

Se ha cogido el canal de color rojo como se podría haber cogido el canal de color verde, simplemente la idea es ilustrar que una imagen que está en escala de grises es una imagen que contiene la misma información en el canal rojo, en el canal verde, y en el canal azul de cada píxel. Otra forma de haber resuelto este ejercicio podría consistir en realizar un promedio entre los tres valores, siempre acordándonos de realizar un redondeo después de realizar el promedio, ya que la información de la cantidad de color de cada píxel debe ser un número entero y sin decimales.

```
from PIL import Image

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])

altura = imagen.size[1]
anchura = imagen.size[0]
```

```
for x in range(0,anchura):
    for y in range(0,altura):
        rojo = pixeles[x,y][0]
        verde = pixeles[x,y][1]
        azul = pixeles[x,y][2]

        rojo = rojo
        verde = rojo
        azul = rojo

        pixeles[x,y] = (rojo,verde,azul)



imagen.save("josevicenteguardado.png")
```

Ejecutar este código no devolverá necesariamente ningún resultado en la pantalla, sino que, al finalizar la ejecución del programa, deberemos abrir la imagen resultante con cualquier editor o visor de imágenes que esté instalado en nuestro sistema, para comprobar que la imagen resultante será una versión en blanco y negro de la imagen original.



13.12.5. Recorrer píxeles grises - método 2

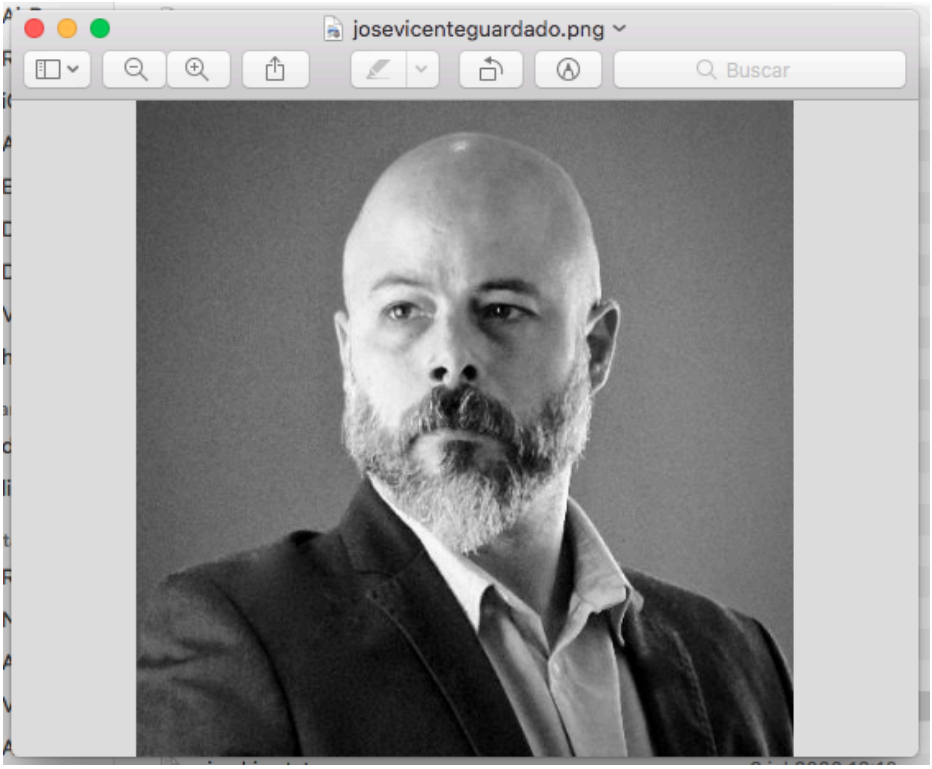
El método descrito en el ejercicio anterior, se pone en práctica dentro de este ejercicio, donde veremos que la diferencia consiste en que creamos una variable llamada color, en la cual realizamos un promedio utilizando la sencilla fórmula de sumar el valor numérico entero de cada uno de los tres canales, dividirlo entre el número tres, y a continuación realizar un redondeo a la baja utilizando Una de las fórmulas de redondeo incluidas dentro de la librería matemática.

De esta forma, estamos realizando un promedio algo más preciso y más exacto que simplemente volcar a la información del canal rojo en los otros dos canales de color.

Si comparamos la imagen generada en este bloque del ejercicio, con la imagen generada en el anterior, comprobaremos que el resultado de convertir la imagen en escala de grises, tiene algo más de calidad y es más respetuoso con respecto al color original de la imagen.

```
from PIL import Image
import math

imagen = Image.open("josevicente.jpg")
pixeles = imagen.load()
print(imagen.size)

print(pixeles[0,0])

altura = imagen.size[1]
anchura = imagen.size[0]

for x in range(0,anchura):
    for y in range(0,altura):
        rojo = pixeles[x,y][0]
        verde = pixeles[x,y][1]
        azul = pixeles[x,y][2]
        color = math.floor((rojo+verde+azul)/3)
        rojo = color
        verde = color
        azul = color

        pixeles[x,y] = (rojo,verde,azul)



imagen.save("josevicenteguardado.png")
```

Al ejecutar este código, Al igual que en ejercicios anteriores, no obtendremos necesariamente un resultado en pantalla, sino que deberemos abrir la imagen resultante para comprobar el efecto de haber pasado la imagen original a escala de grises, en este caso con un algoritmo más respetuoso con los canales verde y azul de la imagen original



14. Epílogo

Si este libro ha cumplido la función que se pretende, al finalizarlo cuentas con muchos más conocimientos acerca de cómo construir programas informáticos en el lenguaje Python de los que tenías cuando lo has empezado a leer.

En función de tus conocimientos de partida, de tu perfil, de tu tiempo disponible, de tu capacidad creativa, y de otros muchos factores, aprovecharás este libro en mayor o menor medida, y te será de utilidad en mayor o menor medida. Yo espero que todo aquello que contiene este libro te sea de la máxima utilidad, porque ese es el motivo por el cual lo he escrito.

¿Cuáles son tus motivos para aprender a programar? En ocasiones hay motivos de estudios, en otras ocasiones hay motivos relacionados con el progreso laboral o la construcción de una carrera profesional, o la ampliación de las capacidades en el puesto de trabajo. En otras ocasiones las personas que aprenden a programar lo hacen buscando un medio de dar salida a sus ideas o proyectos, y en otras ocasiones lo aprenden por afición o autorrealización. Todos esos motivos son totalmente válidos y respetables. Todos esos motivos, al final, tienen que ver con la consecución de una satisfacción personal, en un plano o en otro de tu vida.

Cuando aprendes a programar, tu objetivo es no cometer errores y que tus pequeños programas funcionen correctamente. Te frustras cuando los programas dan error, y te alegras cuando funcionan. Más adelante, tu objetivo consiste en crear programas cada vez más complejos, y que estén diseñados por ti. Poco a poco vas creando programas más grandes, y si funcionan, se convierten en éxitos personales. Más adelante, las personas, los usuarios, empiezan a usar tus programas, y la medida del éxito ya no es si el programa funciona, sino si la gente lo usa.

Los programas informáticos son herramientas en manos de las personas. Les ayudan a realizar tareas que antes no podían hacer, o les permiten hacer tareas de una forma más productiva con respecto a cómo lo hacían antes. Las herramientas ayudan a construir cosas, y las herramientas informáticas, en el mundo actual, ponen mucho poder en manos de las personas. Las personas que usan las herramientas tienen poder, y las personas que crean las herramientas tienen más poder todavía. Aprendiendo a programar, tú eres una de esas personas. Por lo tanto, el fondo de este libro no trata

realmente de variables, de estructuras de control, de funciones, de bases de datos o de programación orientada a objetos. Realmente trata de proporcionar al lector el conocimiento necesario para ser una de las personas que creen herramientas que cambien este mundo - pero te pido que ese cambio sea a mejor.

Piensa en el mundo en el que vives, y piensa en el futuro en el que te gustaría vivir. Y ahora, piensa en qué es lo que hace falta para conseguir vivir en ese futuro mundo ideal que tienes en mente. Por supuesto que un solo programa escrito en Python no va a hacer ese futuro realidad, tienen que ser muchas acciones de muchas personas quienes consigan ese cambio.

Aprendiendo a programar, aprendiendo a desarrollar herramientas, tienes la posibilidad, en el momento en el que las herramientas informáticas cambian el mundo, de participar en ese proceso de cambio. Un solo programa informático no cambiará el mundo. Los programas informáticos no cambiarán el mundo. Para cambiar el mundo a mejor se requiere tener visión y voluntad, pero también se requieren, en ocasiones, herramientas para hacer realidad esa visión. Y eso es lo que nos proporcionan los lenguajes de programación: herramientas.

Ahora depende de ti para qué propósito uses esas herramientas.