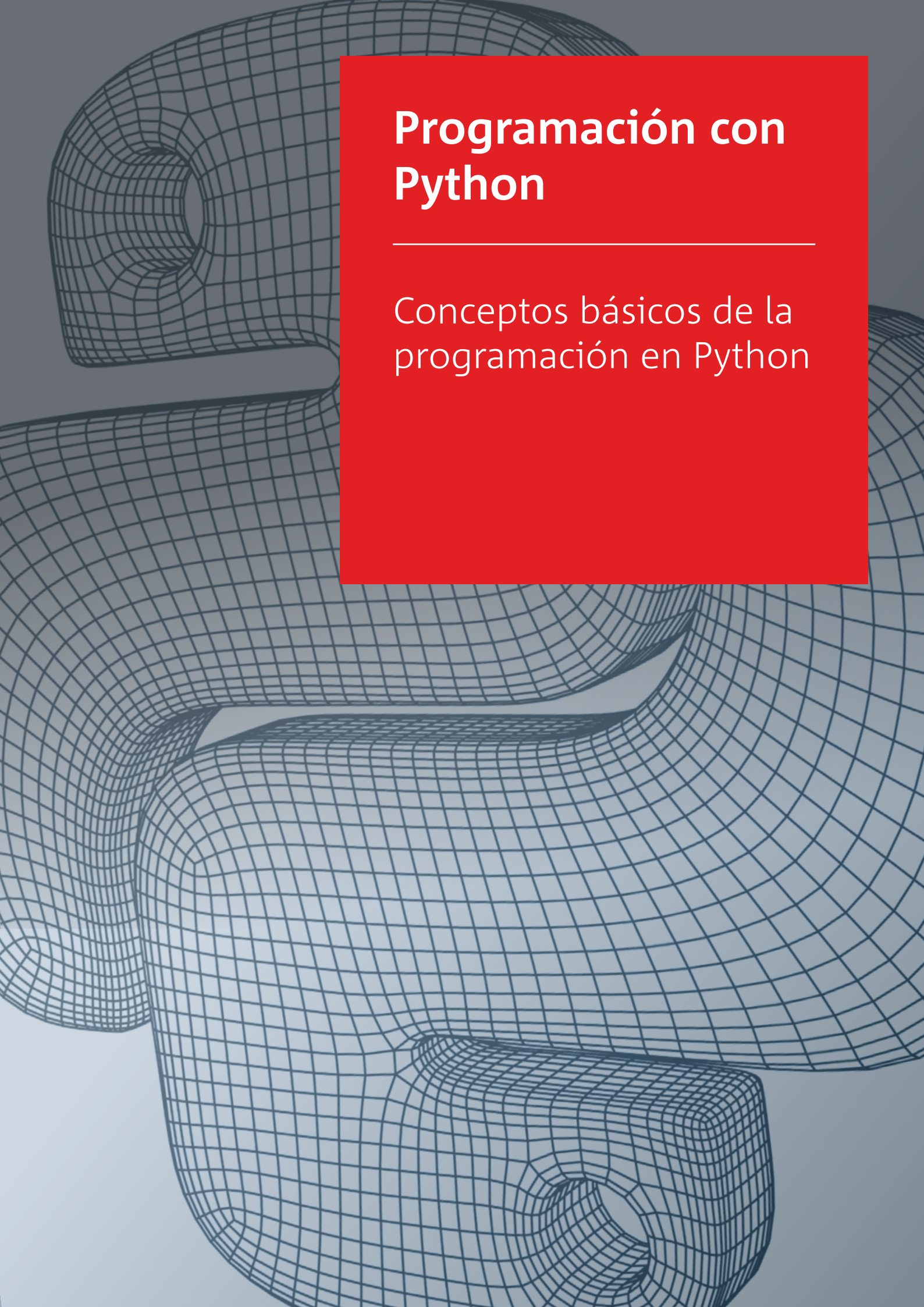

Programación con Python

Conceptos básicos de la programación en Python

INICIATIVA Y COORDINACIÓN
DEUSTO FORMACIÓN

COLABORADORES
Realización:
E-Mafe E-Learning Solutions S.L.

Elaboración de contenidos:
Claudio García Martorell
Licenciado IT Telecomunicaciones especialidad Telemática.
Postgrado en Sistemas de Comunicación y Certificación en Business Intelligence TargIT University.
Concejal de Innovación y Tecnología.
Ponente y docente en distintas universidades y eventos.
Josep Estarlich Pau
Técnico de Ingeniería Informática.
Director Área de Software de la empresa Dismuntel.
Participante en proyectos con Python, C#, R y PHP orientados a *Machine Learning* y a la Inteligencia Artificial.

Supervisión técnica y pedagógica:
Grufium educación y excelencia

Coordinación editorial:
Grufium educación y excelencia

Barcelona (España), 2021

Primera edición: septiembre 2021

ISBN: 978-84-1300-688-8 (Obra completa)
ISBN: 978-84-1300-689-5 (Conceptos básicos de la programación en Python)

Depósito Legal: B 11038-2021

Impreso por:
SERVINFORM
Avenida de los Premios Nobel, 37
Polígono Casablanca
28850 Torrejón de Ardoz (Madrid)

Printed in Spain
Impreso en España

Esquema de contenido

1. Fundamentos de la programación de alto nivel

1.1. Tipos de lenguajes de alto nivel
1.2. Características de los lenguajes de alto nivel
1.3. Implantación de los lenguajes de alto nivel

2. Diferencias entre compiladores e intérpretes

2.1. ¿Qué es un intérprete?
2.2. ¿Qué es un compilador?
2.3. Diferencias entre lenguaje interpretado y compilado
2.4. ¿Cómo funciona el intérprete en Python?

3. Diferencias entre Python 2 y Python 3

3.1. Identificación de versiones en Python
3.2. ¿Qué es Python 2?
3.3. ¿Qué es Python 3?
3.4. Soporte y migración de Python 2
3.5. Principales diferencias entre Python 2 y Python 3

4. Introducción al control de flujo y entradas/salidas de datos

4.1. Palabras reservadas
4.2. Líneas y espacios
4.3. Comentarios y citas
4.4. Operadores
4.5. Delimitadores
4.6. Identificadores
4.7. Funciones integradas y adicionales
4.8. Estructuras de control de flujo condicionales
4.9. Estructuras de control de flujo iterativas

5. Errores frecuentes y depuración de código

5.1. Mala indentación
5.2. Llamada errónea de variables
5.3. Usar palabras reservadas como nombres de variables

5.4. Los espacios como guion bajo
5.5. Usar operando de asignación en lugar de igualdad
5.6. No finalizar correctamente una línea

Introducción

Comenzamos este primer módulo introduciendo conceptos básicos que nos van a dar la información necesaria para identificar perfectamente qué es un lenguaje de alto nivel.

Vamos a ver las características principales, las ventajas y las desventajas de cada uno de los lenguajes de alto nivel más usados hoy en día, como pueden ser Java, PHP, Ruby, JavaScript y, por supuesto, Python.

Dentro de los lenguajes de alto nivel, podremos discernir entre lo que es un lenguaje compilado y uno interpretado, tanto en lo que respecta al proceso de ejecución como al rendimiento.

Alejándonos un poco de los aspectos genéricos de los lenguajes, conoceremos la diferencia entre Python 2 y Python 3, cómo funciona su intérprete y la gestión de los bloques funcionales dentro de un programa desarrollado en Python.

Por último, veremos los errores más comunes a la hora de empezar a programar con Python y cómo podremos depurar errores e intentar escribir un código lo más claro y elegante posible.

1. Fundamentos de la programación de alto nivel

Entre los lenguajes de programación usados hoy en día, se puede hacer una diferenciación atendiendo a criterios de cercanía de su sintaxis al lenguaje humano. Decimos que un lenguaje de programación de alto nivel se caracteriza por tener unas instrucciones y una forma de expresar los algoritmos fácilmente identificables por un humano, bien sea por la forma en la que estructura la información, bien por la secuencia de ejecución de sus bloques o por la forma en la que se ejecuta un algoritmo.

Existe una gran variedad de lenguajes, y siguen creándose continuamente. En esta unidad vamos a centrarnos en menos de una decena de ellos con el objetivo de compararlos. Algunos serán más fáciles de aprender y podrán ser utilizados por desarrolladores noveles. Dicha facilidad implica, por lo general, una reducción en la eficiencia al ser ejecutados; otros tendrán un propósito mucho más específico y por ello tendrán un enfoque más centrado en una tarea concreta, lo que facilitará la resolución de problemas bien identificados (Figura 1.1).

Ejercicios resueltos

Practica el **ejercicio 1** con los conocimientos adquiridos hasta esta unidad.

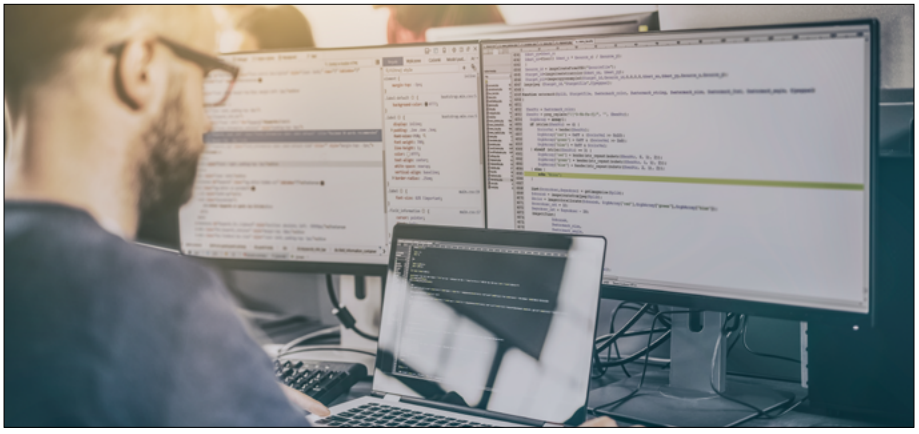

Figura 1.1
Los lenguajes de alto nivel son los más usados y demandados hoy en día.

1.1. Tipos de lenguajes de alto nivel

Los lenguajes de alto nivel, al contrario que el lenguaje máquina o los lenguajes de bajo nivel, consiguen reducir las complejidades para que podamos centrarnos en la necesidad o el problema que necesitamos resolver mediante programación. Esto implica que los lenguajes de alto nivel no pueden ser ejecutados directamente por el *hardware*;

necesitan compiladores e intérpretes para que podamos utilizar todos los recursos. Los más comunes son:

- Estructuras
- Variables
- Matrices
- Cálculos matemáticos complejos o condicionales (bien sean lógicos o aritméticos)

Otras características algo más complejas y típicas de los lenguajes nacidos en los últimos 25 años, sobre todo de los orientados a la interacción con el usuario final y la automatización de procesos, son:

- El uso de funciones
- La orientación a objetos
- La gestión de hilos y semáforos
- La recursividad en los algoritmos

En los lenguajes de bajo nivel, la limitación principal residía en que dependían en buena parte del *hardware* sobre el que estaban ejecutándose. Eso hacía que hubiera tantos lenguajes como fabricantes, que el código no fuese reciclable ni funcional de un aparato a otro y que, por la limitación propia de la tecnología del momento, no pudieran abarcar ni muchos cálculos ni mucha memoria, cosa que hacía que se centrasen en un área o una tarea en concreto (lenguajes lógicos, lenguajes matemáticos, etc.). Tampoco debemos pasar por alto que, obviamente, su sintaxis no era nada visual ni sencilla de comprender si el programador no tenía una formación específica en ese lenguaje.

Hoy en día un programador acostumbrado a trabajar con lenguajes de alto nivel (por ejemplo, PHP), puede hacerse una idea muy aproximada de la funcionalidad para la que se ha diseñado el programa simplemente analizando el código en otro lenguaje de alto nivel (por ejemplo, Java).

A continuación, detallaremos los lenguajes de alto nivel más utilizados en la actualidad (Figura 1.2).

Figura 1.2
Casi todos los lenguajes de alto nivel comparten características.

1.1.1. PHP

- **Año de aparición:** 1994

- **Objetivo:** Lenguaje de programación de uso general especialmente orientado al desarrollo web.

El código PHP es interpretado y ejecutado en el servidor web. PHP puede ser utilizado en otros ámbitos más allá del desarrollo web, como el control de drones, los algoritmos de inteligencia artificial o la creación de interfaces gráficas.

Es un *software* libre publicado bajo licencia PHP. La gran comunidad detrás de este lenguaje ha hecho posible su portabilidad a multitud de plataformas y está aceptado en la gran mayoría de servidores web que existen hoy en día, si bien es uno de los lenguajes en los que las actualizaciones y versiones de seguridad deben tenerse muy en cuenta por la cantidad de ataques recibidos y vulnerabilidades que presenta. Veamos un ejemplo de "Hola Mundo" en PHP:

```
<?php echo '¡Hola, mundo!'; ?>
```

1.1.2. JAVA

- **Año de aparición:** 1995

- **Objetivo:** Java está en todas partes. Inicialmente se creó para controlar las automatizaciones de los estados de cierto tipo de maquinaria (por

ejemplo, una lavadora), pero hoy se encuentra presente en el desarrollo de webs, *smartphones*, juegos, etc.

Es un lenguaje de programación con una orientación a objetos muy marcada y cuyo objetivo es su uso en Internet; no en vano, los navegadores más usuales han incorporado un intérprete de Java en su código.

Este lenguaje de programación fue creado por la compañía Sun Microsystems. En 2007 licenció la mayoría de sus desarrollos de Java bajo la licencia Pública General de GNU. En 2010 Sun Microsystems fue adquirida por Oracle, que se hizo así responsable del mantenimiento del lenguaje Java y de todos sus componentes.

La idea inicial en la creación de este lenguaje tenía como premisa *Write Once, Run Anywhere* ("escríbelo una vez, ejecútalo en cualquier lugar"), dando a entender que la filosofía de Java residía en un código único e independiente de plataforma. Los programas desarrollados en Java son compilados y luego interpretados por su máquina virtual (la *Java Virtual Machine*, JVM). Esta sirve como conexión entre la máquina (*hardware*) y la orden (*software*). Veamos un ejemplo de "Hola Mundo" en Java:

```
class HolaMundo
{
    public static void main(String[] args)
    {
        System.out.println("Hola, mundo.");//Opcion 1

        javax.swing.JOptionPane.showMessageDialog(null,"Hola, mundo.");//Opcion 2
    }
}
```

1.1.3. Javascript

- **Año de aparición:** 1997

- **Objetivo:** Dedicado a aumentar el dinamismo en la interfaz humano-máquina en sitios web.

Se ejecuta principalmente del lado del cliente, ya que está integrado dentro de los navegadores web; sin embargo, también existe el Server-side JavaScript (o SSJS).

Es un lenguaje de programación interpretado y derivado del estándar ECMAScript. Es dinámico y orientado a objetos. Guarda similitudes con Java, ya que hereda nombres y procedimientos, aunque ambos lenguajes tienen sintaxis y propósitos diferentes.

Todos los navegadores actuales interpretan el código JavaScript que está codificado en las páginas web. El hecho de que JavaScript se pueda ejecutar con garantías dentro de una web se debe a que se dota al lenguaje de una implementación del Document Object Model (DOM). Veamos un ejemplo de "Hola Mundo en Javascript:

```
//Caso write()
document.write("¡Hola, mundo!");

//Caso alert()
alert("¡Hola, mundo!");

//Caso console.log()
console.log("¡Hola, mundo!");
```

1.1.4. C#

- **Año de aparición:** 1999

- **Objetivo:** Realización de aplicaciones web, móviles y de escritorio. Está estandarizado como parte de la plataforma .NET de Microsoft; no en vano, ha sido esta empresa la que lo ha estandarizado.

Es un lenguaje multiparadigma que hereda gran parte de su *core* de C/C++. En el tratamiento de objetos, asimila la misma manera de proceder que Java. Si bien es un lenguaje derivado de C/C++, elimina ciertos atributos redundantes y objetos para favorecer que la sintaxis sea más intuitiva, y el código, menos pesado.

Un aspecto muy importante que hay que destacar es la seguridad, ya que tiene mecanismos robustos para asegurar los objetos, tanto en su creación como en su tratamiento.

Por otro lado, tiene un sistema de tipos unificados, cosa que juega en favor de su extensibilidad, puesto que se puede añadir otro tipo de datos que se consideren básicos, modificadores y operadores.

Pese a que la tasa de adopción de C# es estable entre la comunidad de programadores, es integrable con multitud de lenguajes de alto nivel, lo que lo posiciona como una de las alternativas de lenguaje base para desarrollar un programa y poder ir incluyendo funciones, conexiones o integraciones con otros lenguajes. Veamos un ejemplo de "Hola Mundo" en C#:

```
using System;

 class MainClass
 {
     public static void Main()
     {
         Console.WriteLine("¡Hola, mundo!");
     }
 }
```

1.1.5. Ruby

- **Año de aparición:** 1993

- **Objetivo:** Crear aplicaciones web y de escritorio.

Es un lenguaje de programación interpretado que se basa principalmente en la orientación a objetos. Desde ese punto de vista, todo es un objeto y, por tanto, se le puede aplicar herencia, métodos, etc.

Es rápido en la ejecución y sencillo en la sintaxis. Al igual que muchos otros lenguajes de alto nivel, sus variables no necesitan ser declaradas (lenguaje dinámico). Aun así, las variables sí conservan el tipo con el que se inicializan.

Igual que Java, tiene un recolector de basura o gestión dinámica de memoria en segundo plano. Soporta la programación de hilos y cuenta con gestión de excepciones. Guarda muchas similitudes en la sintaxis y el tratamiento de los algoritmos con Python y Perl.

Al ser un lenguaje *open source*, libre y multiplataforma, cuenta con una gran comunidad detrás y muchas librerías ya predefinidas para ayudar al programador. Veamos un ejemplo de "Hola Mundo" en Ruby:

```
puts "Hola, mundo."
```

1.1.6. Python

- **Año de aparición:** 1991

- **Objetivo:** Inicialmente, su uso más extendido fue para realizar páginas web dinámicas, aunque hoy en día es el lenguaje principal para cualquier programa basado en *data science*, *big data* o inteligencia artificial.

Fue creado por Guido van Rossum como alternativa a Perl en un primer momento, pero mejorando la claridad de la sintaxis y con un nivel de abstracción mucho mayor, cosa que ayuda a la mejor comprensión por parte del programador. El nombre del lenguaje, contra lo que podría parecer (más aún a la vista del logotipo), no tiene ninguna connotación animal o en referencia a las serpientes pitón (*python*, en inglés), ya que proviene de la afición de su creador por el grupo de humor británico Monty Python.

Está gestionado por la Python Software Foundation, que lo tiene licenciado bajo una licencia de código abierto llamada Python Software Foundation License.

Este es un lenguaje interpretado y multiparadigma, además de multiplataforma y dinámico. Esto presenta las ventajas ya conocidas respecto a los lenguajes compilados, y sus principales características son la velocidad y la reducida cantidad de líneas necesaria para desarrollar una función.

Tiene un intérprete incorporado que puede utilizarse de forma interactiva, lo cual favorece mucho el aprendizaje, ya que permite ir experimentando poco a poco con las características propias del lenguaje.

Otra de las características distintivas de Python es la modularidad, ya que podemos partir o compartimentalizar nuestro código en pequeños módulos que podrán ser accesibles desde otros programas hechos en Python; no en vano, viene con módulos ya predefinidos a los que se puede acceder para la ejecución de funciones concretas, como la entrada/salida de ficheros, sockets, llamadas, etc. Python destaca además por gestionar una resolución dinámica de nombres; es decir, el vínculo entre un método y una variable durante el tiempo de ejecución. Veamos un ejemplo de "Hola Mundo" en Python:

```
print ("Hola, mundo.")
```

1.2. Características de los lenguajes de alto nivel

Antes de destacar las ventajas y desventajas de los lenguajes de alto nivel para los procesos de creación de *software* actuales, y teniendo en cuenta los ejemplos de código "Hola Mundo" listados anteriormente de cada lenguaje, ya se puede apreciar qué lenguajes son más intuitivos a la hora de escribir las órdenes, o cuáles ocupan más líneas de código para ejecutar una instrucción tan simple.

Llegados a este punto, hay que hacer referencia a un paper de la Universidad de Karlsruhe titulado *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program* y escrito por Lutz Prechelt, donde se comparaban varios aspectos que hay que tener en cuenta de diferentes lenguajes, atendiendo a criterios de eficiencia, carga en memoria, rapidez de escritura, cantidad de código, etc. para una misma tarea en forma de algoritmo.

Las conclusiones, a modo de *benchmarking*, son muy interesantes desde el punto de vista de la comparativa de lenguajes compilados como C y C++, lenguajes interpretados como Python o Perl, y Java, que se puede considerar como híbrido:

- En lo que respecta al tiempo de desarrollo, se tardaba en torno a un 40% menos en escribir el mismo código en Perl y Python que en Java, C o C++.
- La mejor gestión de la memoria la tenían C y C++, mientras que Python, Perl y Java requerían el doble de consumo de posiciones de memoria.
- En lo que respecta a la búsqueda y comparación de datos, Python era el más eficiente, seguido de Perl; C y C++ requerían más del doble de tiempo.

Certificación

Es importante interiorizar las características de los lenguajes de alto nivel, pues en numerosas preguntas de la certificación se hace referencia a estas.

Este tipo de experimentos y comparativas nos pueden dar una pista, pero no una certeza, sobre las prestaciones de un lenguaje de programación. Un mismo problema se puede afrontar de manera diferente con cada lenguaje, pero son mucho más importantes la pericia del programador y su dominio del lenguaje, ya que el mismo algoritmo se puede expresar de diversas maneras, más o menos eficientes.

Otro factor que hay que tener en cuenta es el objetivo del algoritmo, ya que hay lenguajes más optimizados para una serie de cálculos que otros. Aunque se pudiera hacer, no tendría sentido intentar ser muy

eficiente gestionando contenido de vídeo online en directo con un lenguaje que está pensado para optimizar comunicaciones entre dispositivos de internet de las cosas (IoT).

Si bien los lenguajes de alto nivel han desplazado a los de bajo nivel para la gran mayoría de desarrollos tecnológicos de hoy en día, no hay que negar la evidencia de que algunos de los lenguajes de bajo nivel siguen y seguirán utilizándose por ser muy específicos y concretos para algunas tareas.

Pasemos a analizar qué ventajas o inconvenientes pueden representar los lenguajes de alto nivel una vez que nos decantamos por la programación en alguno de ellos.

1.2.1. Ventajas

Las ventajas del lenguaje de alto nivel son las siguientes:

- Curva de aprendizaje muy suave para nuevos programadores debido a la proximidad al lenguaje humano. Además, el tiempo de formación de un programador para que sea mínimamente funcional es bastante más corto en comparación con un programador que se tenga que formar en lenguajes de bajo nivel.

- La modificación del código es mucho más sencilla, ya que se usan funciones y objetos, por lo que no hay que editar todo el código, sino solo la parte que nos interesa.

- Sintácticamente, los comandos suelen ser instrucciones cuyos nombres son palabras conocidas y que nos dan una información completa sobre qué van a hacer: *print, read, open, destroy, write*.

- Sencillez y rapidez de uso. Los procesos que antes requerían una gran inversión de recursos se pueden llevar a cabo de forma inmediata con este tipoo de lenguaje.

- El código se puede reutilizar y ser llamado desde varios puntos del programa. Además, se podrá ejecutar en varios sistemas operativos y plataformas. Esta independencia del *hardware*, sumada a la polivalencia que pueda tener el código, hace que el uso de este tipo de lenguajes se extienda cada día más.

- Se pueden usar los paradigmas de programación.

- El lenguaje recomienda y empuja a que continuemos con el proceso de práctica a fin de dar forma a programas que podrán ser realmente avanzados.

- Se utilizan muchas menos líneas de código.

1.2.2. Desventajas

Las desventajas del lenguaje de alto nivel son las siguientes:

- Las cualidades propias del *hardware* no representan un factor diferencial, puesto que el programa es independiente de la plataforma, y se infrautilizan los recursos internos de la máquina.

- Se hacen consultas más pesadas y algoritmos más complejos, que dan como resultado un tiempo de ejecución muchísimo mayor.

- Derivada del punto anterior, otra desventaja reside en que las máquinas tendrán que estar optimizadas de alguna forma para poder ejecutar los programas desarrollados con lenguajes de alto nivel. Es posible que se tengan que instalar plataformas de configuración, compiladores o intérpretes; y, por otro lado, si el *hardware* y la red de comunicaciones no están dimensionados para dar un servicio óptimo, se ocasionarán retrasos a la hora de obtener la información.

Aun así, es obvio que hoy en día, y con la evolución que tiene la tecnología, las ventajas pesan más que las desventajas. No en vano, se puede observar de manera muy clara la evolución y la penetración de cada lenguaje en la comunidad desarrolladora, sobre todo en el ámbito laboral.

1.3. Implantación de los lenguajes de alto nivel

El foro StackOverflow pasa por ser uno de los sitios web de referencia más famosos y que más tráfico de consultas sobre lenguajes de programación aglutinan. Es una comunidad que formula preguntas, propone soluciones e incluso reporta vulnerabilidades que son tenidas en cuenta por las propias empresas o fundaciones encargadas del mantenimiento de algunos lenguajes de programación. Cada año, esta organización hace público un reporte sobre tendencias basadas en las búsquedas que se realizan en su sitio web, estimando así qué es lo más demandado en cada momento.

La demanda de información sobre Python en los últimos años se ha disparado, dejando atrás a lenguajes mucho más arraigados y populares (Figura 1.3).

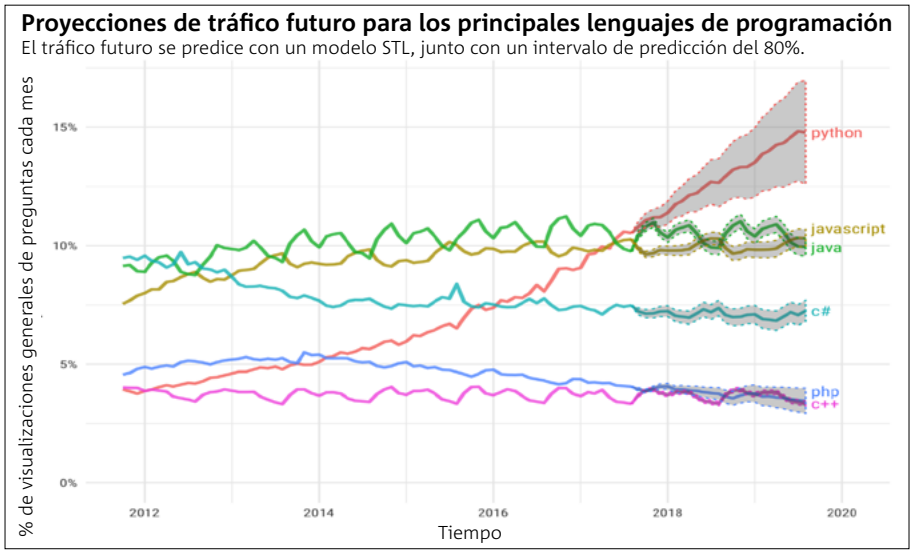

Figura 1.3
Aumento de las consultas sobre lenguajes de programación desde 2012 en Stack Overflow.

Haciendo una búsqueda independiente mediante la plataforma Google Trends sobre los términos más buscados en relación con algunos de los lenguajes de alto nivel más importantes, podemos observar una tendencia muy parecida a la descrita por las búsquedas realizadas en StackOverflow (Figura 1.4).

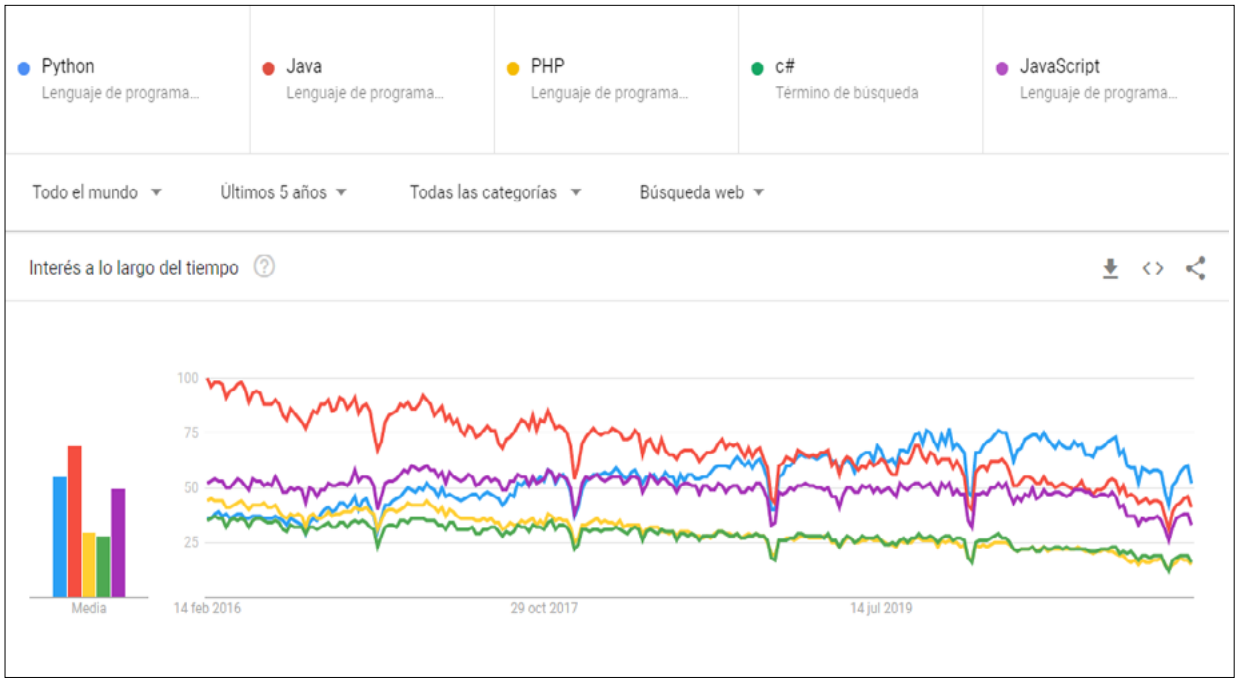

Figura 1.4
Evolución de las búsquedas en Google por nombre de lenguaje de programación.

Siguiendo con esta metodología de búsqueda, pero atendiendo a criterios geográficos, obtenemos un mapa bastante visual que nos da la dimensión adecuada de por qué Python está creciendo tanto en los últimos cinco años tal y como nos muestra el mapa geográfico (Figura 1.5).

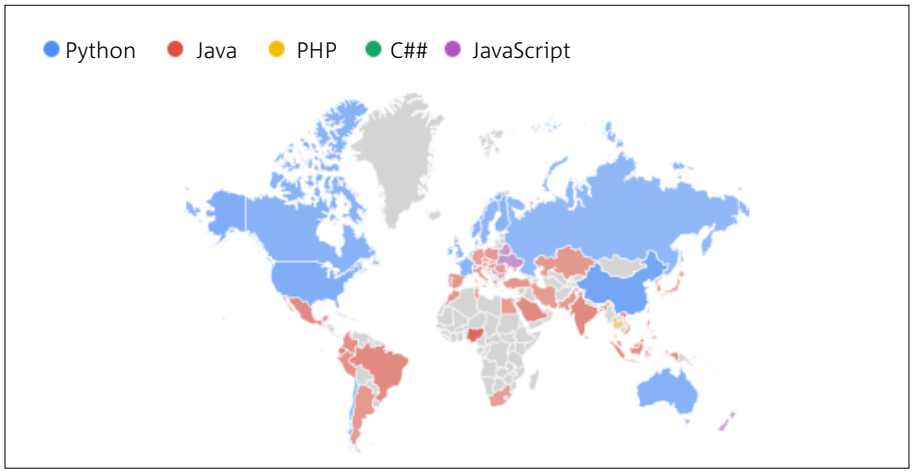

Figura 1.5
Distribución geográfica mundial de las búsquedas por lenguaje de programación.

Y es que, siendo uno de los lenguajes con menos representación hace poco menos de una década, hoy en día es el más buscado en la mayoría de los países, incluyendo potencias tecnológicas como China, Rusia o Estados Unidos.

Resumen

- Los lenguajes de alto nivel son una evolución de los lenguajes máquina o de bajo nivel. Si bien permiten programar con más fluidez y su sintaxis se acerca cada vez más a la sintaxis humana, tienen las desventajas de no estar optimizados al 100% para el *hardware* sobre el que se ejecutan.

- Los lenguajes de bajo nivel apenas se usan hoy en día porque están ligados al *hardware* sobre el que son ejecutados; eso les confiere una gran especialización, pero muy poca o nula flexibilidad a la hora de ejecutarse en otros sitios, por lo que el código difícilmente se puede reaprovechar.

- Python es el lenguaje de alto nivel que más está creciendo en los últimos cinco años de manera consecutiva, ya que está siendo el más adoptado por la comunidad de desarrolladores, sobre todo en los campos de *big data* o *machine learning*.

2. Diferencias entre compiladores e intérpretes

Los lenguajes de alto nivel pueden ser interpretados o compilados. Para hacer la traducción entre código y lenguaje máquina, hará falta un intérprete en el caso de los lenguajes interpretados, mientras que en el de los compilados deberemos tener un compilador.

En esta unidad, vamos a analizar, desde el punto de vista del rendimiento, la economía del lenguaje y la dificultad, las características de los lenguajes interpretados y los compilados, así como sus ventajas e inconvenientes.

Conoceremos también los ejemplos más representativos de cada tipo de lenguaje y, por otra parte, se analizará el funcionamiento del intérprete de Python, así como su ciclo de ejecución.

Para saber más

El código fuente está conformado por líneas de texto con instrucciones escritas en un lenguaje de programación diferente al lenguaje máquina, por lo que es necesario traducirlas para que la máquina pueda ejecutarlas.

2.1. ¿Qué es un intérprete?

Un **intérprete** es un programa que interpreta un **código fuente**, ejecuta a partir de este una serie de acciones y puede llamar o ejecutar otros programas. Esta ejecución se va realizado a medida que el intérprete traduce el código, línea a línea. Uno de los puntos más importantes es que el intérprete simplemente ejecuta lo que lee y traduce, pero no guarda la traducción (Figura 2.1).

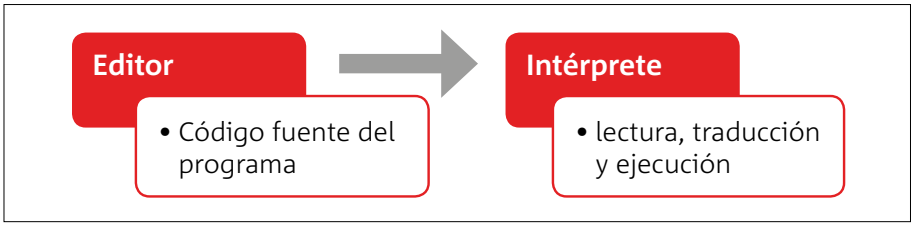

Figura 2.1
Fases de ejecución de un lenguaje interpretado desde el punto de vista del programa implicado.

Existen básicamente dos tipos diferentes de intérpretes: los **incrementales** y los **avanzados**.

- **Intérpretes incrementales:** se usan para ejecutar y dar respuesta en sistemas interactivos, donde ya se pueden encontrar módulos compilados y módulos que se pueden modificar.

- **Intérpretes avanzados:** sin llegar a ser compiladores en sentido estricto, estos intérpretes permiten realizar un análisis muy pormenorizado del código fuente y además integran un análisis previo (de seguridad, comprobación de estructura, etc.) del programa que está siendo creado.

Se podría considerar un tercer tipo, los **intérpretes híbridos**, que utilizarían procesos y metodologías usados por los compiladores, sumando además la estructura de traducción de los intérpretes.

2.2. ¿Qué es un compilador?

Un compilador es un programa que traduce el código que nosotros escribimos en cualquier lenguaje compilado y lo traduce a código máquina ejecutable. La compilación se producirá si no hay ningún error en el código; en caso de haberlo, el compilador mostrará en pantalla un listado de errores o advertencias que hemos de tener en cuenta para que nuestro código sea 100% convertible a ejecutable. El compilador no ejecuta, simplemente encapsula todo el código en un programa que deberemos ejecutar en un segundo paso.

El proceso de compilación pasa por tres fases diferentes, y en cada una de ellas se genera un fichero y es necesaria la acción de un programa concreto. Atendiendo al tipo de fichero generado en cada fase del proceso, tendríamos el siguiente diagrama (Figura 2.2).

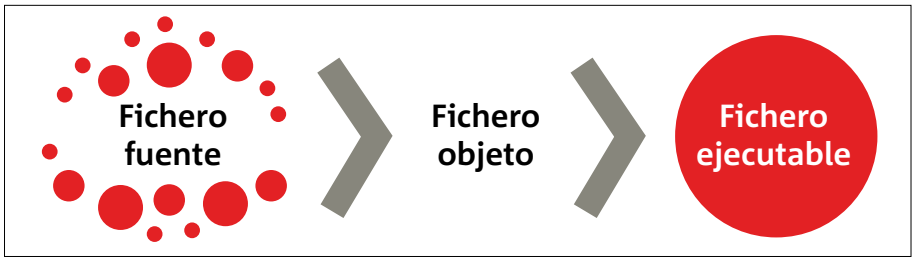

Figura 2.2
Fases de ejecución de un lenguaje compilado desde el punto de vista de los ficheros generados.

Se parte del fichero fuente, donde está el código de nuestro programa. Sobre este fichero, el compilador va a realizar tres tipos de análisis para comprobar la idoneidad del código:

- **Análisis léxico:** se parte del código o fichero fuente, se segmentan todas las órdenes (creando *tokens*) y se eliminan los comentarios del programador y los espacios en blanco. Si se encuentra algún token inválido, se genera un error.

- **Análisis sintáctico:** se toman los *tokens* ya validados léxicamente y se analizan a partir de una serie de reglas gramaticales propias del lenguaje en el que se esté desarrollando. Convierten el texto de entrada en una estructura de árbol, jerarquizando así los *tokens* y dándoles una estructura de datos.

- **Análisis semántico:** se revisa el árbol y la jerarquía producida en el análisis anterior desde el punto de vista del tipo de dato, los operadores, las proposiciones, etc. Es de vital importancia la verificación de tipos, que consiste en comprobar que cada dato tiene los operadores permitidos para su tipo según las especificaciones del lenguaje.

2.3. Diferencias entre lenguaje interpretado y compilado

Una vez que esos tres análisis son favorables y no hay ningún error, el compilador genera el **fichero objeto**. Este fichero objeto será a su vez cargado y completado por un programa enlazador o *linker*, que incluirá las **bibliotecas** que se hayan especificado en el código fuente. Esta inclusión dará como resultado un fichero ejecutable.

Desde el punto de vista del programa usado en cada fase, la secuencia sería tal como se indica en el esquema (Figura 2.3).

Para saber más

Una biblioteca o librería es un conjunto de funciones y métodos previamente traducidos a código máquina, y que sirve principalmente para ayudar al programador a crear *software*.

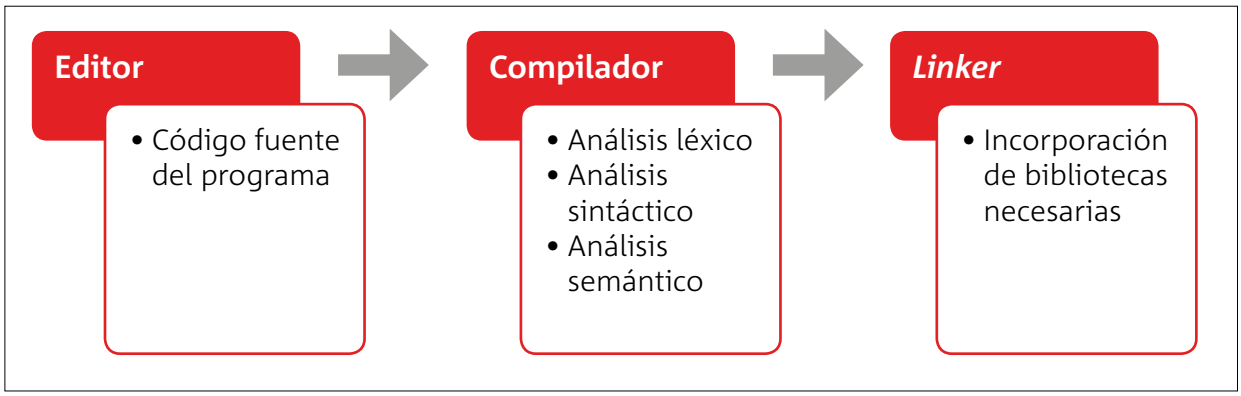

Figura 2.3
Fases de ejecución de un lenguaje compilado desde el punto de vista del programa implicado.

El tratamiento que hace cada lenguaje de su código, dependiendo de si es interpretado o compilado, se podría resumir de la siguiente manera:

- **Interpretado:** cuando el código de un lenguaje es interpretado, sus algoritmos son leídos y ejecutados en el momento, sin ninguna fase intermedia de compilación. Esto se hace gracias a un programa llama-

do intérprete que lee secuencialmente todas y cada una de las instrucciones del programa y las ejecuta según el flujo.

- **Compilado:** cuando el código de un lenguaje es compilado, sus algoritmos son transformados en un ejecutable como paso previo a la ejecución del código en sí. En un segundo paso, el usuario o alguna instrucción automatizada hará que el ejecutable obtenido de la compilación se inicie. Hoy en día existen dos tipos de compilación:

  - **La generación de código máquina:** se produce cuando el compilador traduce el código fuente del programador a código máquina de forma directa.

  - **La representación intermedia:** se produce cuando se compila una parte del programa optimizada para su ejecución posterior, sin la necesidad de volver a leer el código fuente. La forma más común de guardar este tipo de información es en formato de *bytecode*. Este es el caso concreto de Java.

La principal diferencia entre ambos tipos de lenguaje estriba en el comportamiento a la hora de ejecutarse: mientras que, en un caso, el código necesita ser compilado para después ejecutarse, los lenguajes interpretados se saltan ese paso, ya que el intérprete va convirtiendo en lenguaje máquina el código a medida que lo lee.

Tanto el compilador como el intérprete convierten el código fuente editado por el programador en código máquina entendible por el ordenador. Dado que el código máquina está unido al *hardware* de la máquina que lo ejecuta, ese código no se puede ejecutar en otras máquinas.

Sin duda, se aprecia una ventaja en cuanto a fluidez y **ciclo de desarrollo** en el caso de los lenguajes interpretados, si bien es cierto que los lenguajes compilados son más completos y con unas instrucciones mucho más complejas que hacen que sus programas tengan, por lo general, unas funcionalidades mucho más completas que los interpretados.

Sin embargo, y atendiendo única y estrictamente a criterios de velocidad de ejecución, aunque pueda parecer que el lenguaje interpretado sea más rápido por saltarse el paso de la compilación, resulta mucho más eficiente tener ese paso de compilación una vez y, a partir de ahí, poder ejecutar el código tantas veces como se quiera con un tipo de procesamiento casi nulo, que tener que interpretar cada vez que se llame a un trozo de código por el intérprete; además, hay que tener presentes los

Certificación

Es importante interiorizar las diferencias entre lenguaje interpretado y compilado, pues en numerosas preguntas de la certificación se hace referencia a estos.

Ejercicios resueltos

Practica el **ejercicio 2** con los conocimientos adquiridos hasta esta unidad.

Para saber más

El ciclo de desarrollo es el tiempo comprendido desde el momento en que se escribe el código fuente del programa hasta el momento en que este se prueba.

intérpretes web, que muchas veces dependen de las velocidades de la línea o de carga y respuesta de la propia página.

El lenguaje compilado desplaza la carga de trabajo hacia el programador, ya que está más optimizado para la ejecución en la máquina; en cambio, los lenguajes interpretados aumentan la carga de trabajo de la máquina, porque apuestan por que el programador tenga mayor flexibilidad a la hora de escribir y modificar el código.

Como ejemplo de lenguajes compilados y su uso principal, tenemos:

- **Basic:** educación y formación.
- **C:** programación de sistemas.
- **C++:** programación de sistemas orientada a objetos.
- **Java:** programación orientada a Internet.
- **Pascal:** educación.
- **Swift:** aplicaciones para iOS.
- **Fortran:** cálculo.
- **C#:** programación de sistemas orientada a objetos.

Como ejemplos de lenguajes interpretados y su uso principal, podemos encontrar:

- **Ruby:** creación de sitios web.
- **Python:** inteligencia artificial y gestión del dato.
- **JavaScript:** interactividad y dinamismo en sitios web.
- **Lisp:** inteligencia artificial.
- **VBScript:** conexión con ASP y aplicaciones para IIS.
- **PHP:** desarrollo de webs dinámicas.
- **PostScript:** datos en archivos gráficos y descripción de páginas.

Es innegable que estos lenguajes de programación de alto nivel se han hecho un hueco en la mayoría de los desarrollos tecnológicos de hoy en día de forma significativa, y arrastran a una comunidad de usuarios y programadores que no hace más que alimentar la creación de nuevas versiones y la adopción de nuevos estándares, sobre todo en cuestión de seguridad.

Esto se debe a que, de un tiempo a esta parte, no solo se ha ido perfeccionando y segmentando el *software* hacia canales y tecnologías más concretos, sino que además la evolución del *hardware* y las comunicaciones han permitido que los lenguajes de alto nivel puedan permitirse el lujo de tener cálculos mucho más complejos y consultas más pesadas que podrían representar tiempos de ejecución o de transmisión grandes, pero hoy en día representan tiempos despreciables.

2.4. ¿Cómo funciona el intérprete en Python?

Existen dos formas de utilizar el intérprete de Python:

- **Modo *shell* o comandos:**

  - El intérprete nos indicará con “>>>” que está pendiente de recibir una instrucción.

  - Es útil para probar trozos nuestro código.

```
>>>
>>> print (‘El estilo de música que más me gusta \ es el
“Rock\’n\’Roll”’)
El estilo de música que más me gusta es el Rock’n’Roll”
>>>
```

- **Modo *script* o programa:**

  - Debemos escribir un programa completo en un archivo con la extensión .py, y para ejecutarlo con el intérprete podemos utilizar la palabra *python*.

  - Todos estos archivos deben contener una línea *shebang* (#!/usr/bin/python), donde también indicamos la ruta del intérprete. Es muy importante dar permisos de ejecución (chmod u+x .py).

Para validar nuestro código, también podemos utilizar intérpretes interactivos *online* con los que podremos testear sin instalar nada; por ejemplo, Google Colab, Replit, Python Tutor o Jupiter (Figura 2.4).

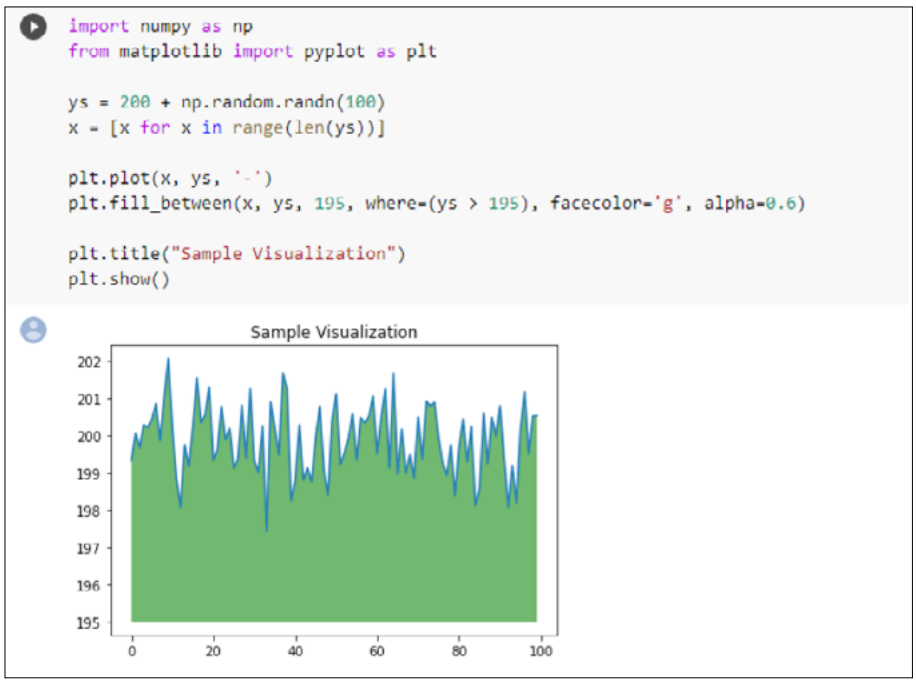

Figura 2.4
Ejemplo de 'script' para la generación de gráficas.

A continuación, vamos a diseccionar un código muy sencillo en Python para ver secuencialmente cómo actúa su intérprete. Como ya sabemos, el intérprete lee línea a línea, pero podría darse el caso de que quisiéramos que Python recordase un valor concreto para hacer cálculos posteriores. Ese valor se guardará en una variable, término que usaremos para referirnos a los datos almacenados. Vamos a pasarle al intérprete un par de cálculos matemáticos sencillos.

```
>>> x = 5
>>> print(x)
5
>>> y = x + 14
>>> print(y)
19
>>>
```

En la primera línea, definimos una variable que se llama *x* y le asignamos un valor numérico, 5. En la segunda línea, comprobamos que el valor ha sido guardado correctamente haciendo que se nos muestre en pantalla. En la tercera línea, recuperamos el valor guardado en la variable *x*, a la que vamos a sumar 14 y vamos a guardarla en otra variable llamada *y*.

Tal como hicimos en la tercera línea, en la quinta vamos a asegurarnos de que el cálculo y el guardado del dato han sido correctos; para ello, hacemos que se nos muestre en pantalla el contenido de la variable *y*.

Como podemos ver, el intérprete toma instrucciones y las ejecuta; su principal misión es mantener una especie de conversación en la que nosotros escribimos línea a línea, pero Python las trata como una secuencia de órdenes cuyas últimas líneas son capaces de recuperar datos creados en las anteriores.

Podemos decir que, en este ejemplo sencillo, el intérprete de Python se comporta como podría hacerlo un *shell* de Linux. En general, el proceso sería más similar al descrito en la ilustración (Figura 2.5).

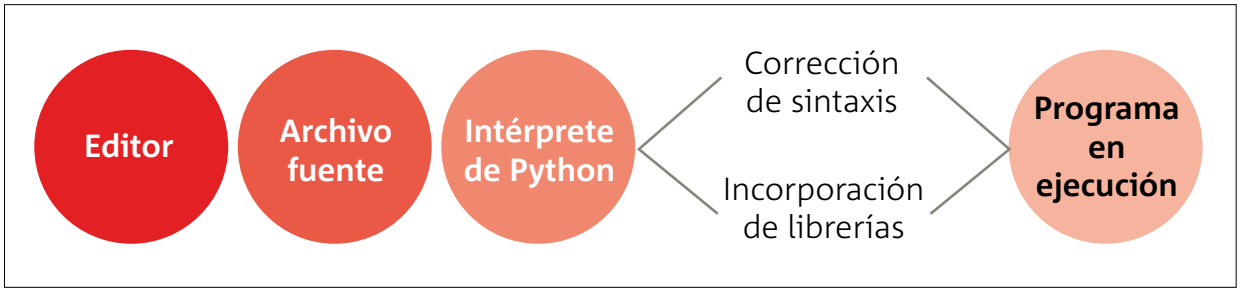

Figura 2.5
Secuencia de ejecución del intérprete de Python.

Resumen

- Un intérprete es un programa que interpreta un código fuente, ejecuta a partir de este una serie de acciones y puede llamar o ejecutar otros programas.

- El compilador es un programa que toma el código que nosotros escribimos en cualquier lenguaje compilado y lo traduce a código máquina ejecutable.

- La elección de un lenguaje compilado o interpretado dependerá del proyecto que vayamos a afrontar y de sus especificaciones. No los hay mejores ni peores por ser interpretados o compilados, simplemente los hay más o menos indicados para el tipo de proyecto.

- Tanto los lenguajes compilados como los interpretados tienen que echar mano en algún punto de su ciclo de ejecución de librerías o bibliotecas de terceros para complementar con funciones, métodos o especificaciones estándar las funcionalidades del programa. Hay bibliotecas con definiciones matemáticas, gráficas, de inteligencia artificial o de tratamiento de imágenes.

3. Diferencias entre Python 2 y Python 3

En los últimos 30 años han existido tres versiones distintas de Python. Actualmente, la Python Software Foundation solo da soporte a la versión Python 3, pero aún quedan sistemas y módulos en activo con la versión 2.7 (la última de Python 2). Por ello, en esta unidad vamos a ver cómo diferenciar ambas versiones, el rendimiento y las características de cada una; así como las directrices para migrar de Python 2 a Python 3 con garantías de éxito.

La gran expansión del lenguaje Python se dio con Python 2, al ser incluido por defecto en distribuciones Linux, por lo que hay que tener en cuenta y analizar en profundidad cuál ha sido su evolución hasta la versión actual, donde podremos conocer los bloques de código más usados, así como analizar los errores más frecuentes a la hora de desarrollar.

3.1. Identificación de versiones en Python

Las versiones de Python (Figura 3.1) se identifican por tres números que siguen el formato **"a.b.c"**:

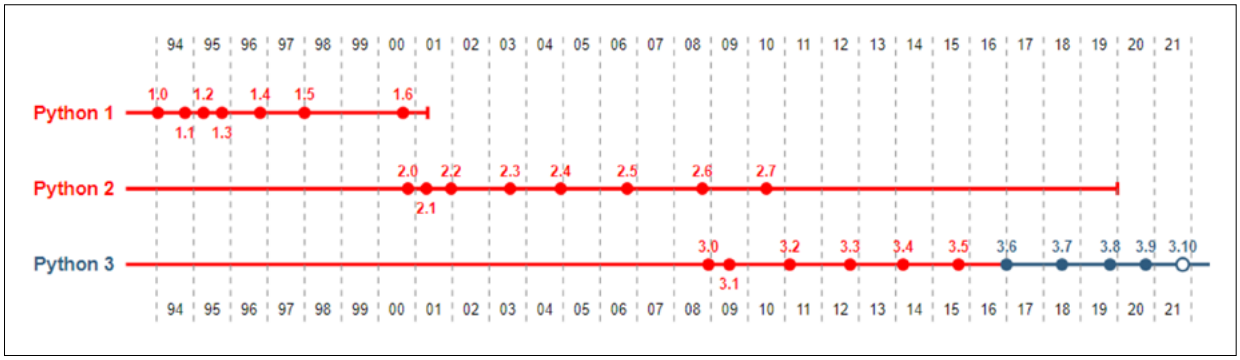

Figura 3.1
Cronología de las versiones de Python.

- **"a"** corresponde a las grandes versiones de Python (1, 2 y 3), incompatibles entre sí.

- **"b"** corresponde a versiones importantes en las que se introducen novedades en el lenguaje, pero manteniendo la compatibilidad (salvo contadas excepciones).

- **"c"** corresponde a versiones menores que se publican durante el período de mantenimiento, en las que solo se corrigen errores y se aplican parches de seguridad si es necesario.

3.2. ¿Qué es Python 2?

Python 2 es una versión descontinuada del lenguaje Python, que apareció por primera vez en octubre del año 2000. La primera versión del lenguaje Python se empezó a desarrollar a finales de los 80 y se lanzó oficialmente en enero de 1994 bajo la denominación Python 1.

Siguiendo un poco el curso de la historia del equipo de desarrollo, en el año 2000 el núcleo principal de programadores de Python se unió a BeOpen para formar el equipo que acabaría llamándose BeOpen Python Labs. Python 2.0 fue el primer (y único) lanzamiento del equipo de desarrollo mientras estuvo aliado con BeOpen.com. Tras la publicación de la versión 2.0, el creador de Python, Guido van Rossum, y todos los demás programadores de lo que denominó Python Labs, se unieron en Digital Creations.

Este cambio no se produjo solo por diferencias de criterio, sino también por seguir con la filosofía que Van Rossum había marcado para el lenguaje Python desde sus inicios; por esa razón, se tendió más hacia un modelo cercano a los modelos de *software* de la Free Software Foundation (FSF). Este movimiento, aunque no se percibió como tal en su momento, acabaría por desembocar en la creación de la licencia Python Software Foundation License. Solo un año después de esto, en 2001, la Python Software Foundation (PSF) pasó a ser dueña del proyecto y de su mantenimiento. Esta fundación se creó a semejanza de la Apache Software Foundation (Figura 3.2).

Figura 3.2
El lenguaje Python está mantenido por la Python Software Foundation

En su momento, el salto a la versión Python 2.0 incluyó varias modificaciones importantes respecto a las características del lenguaje en sus anteriores versiones, empezando a conformar así lo que serían las bases del lenguaje que vería su gran expansión con la versión 2.7, que fue la más extendida, con una duración de 10 años (cuando, por norma, se cambiaba de versión cada 5 años). Algunas de estas características fueron:

- Listas por compresión.
- Sistema de recolección de basura para referencias cíclicas.
- Unificación de tipos de datos escritos en C y de tipos de datos escritos en Python dentro de una misma jerarquía. Esto dio como resultado un modelo de objetos más robusto y consistente.
- Introducción de generadores para el control de los iteradores dentro de un bucle.
- El *parser* SAX.
- Adiciones de bibliotecas.
- Las cadenas Unicode.
- Las asignaciones aumentadas.
- Los nuevos métodos de cadenas.

Su soporte debió finalizar en 2015, pero a causa de su popularidad y la cantidad de distribuciones que lo incorporaban como versión de Python por defecto, la Python Software Foundation decidió alargar el soporte hasta el 1 de enero de 2020.

3.3. ¿Qué es Python 3?

Python 3000, **Py3k** o simplemente **Python 3**, es la versión de Python más actual y, hoy por hoy, la única que tiene soporte por parte de los voluntarios de la Python Software Foundation. Se lanzó en diciembre de 2008.

Van Rossum, el creador de Python, inició el proyecto de Python 3 para limpiar y empezar de cero algunas características de Python 2, ya que

creía que esta era una manera más eficiente de depurar y optimizar código que la metodología de obsoletizar funciones, forzando a la reescritura de métodos, el abandono de clases y la conversión de tipos y funciones. Creó así Python 3, que, debido a esta serie de cambios, no es compatible con Python 2. De haber seguido con parches y cambios menores, la transición de Python 2 a Python 3 hubiese sido mucho menos problemática, pero también mucho menos eficaz.

En palabras de la propia Python Software Foundation, el cambio de modelo y filosofía en Python 3, aunque traumático en algunos casos, era necesario: "Nos dimos cuenta de que necesitábamos hacer grandes cambios para mejorar Python. Muchos usuarios no actualizaron y no queríamos lastimarlos. Por lo tanto, durante muchos años, hemos seguido mejorando y publicando Python 2 y Python 3; pero eso dificulta la tarea de mejorar Python".

Como filosofía del nuevo proyecto Python 3, Van Rossum lanzó tres puntos que hay que tener en cuenta a la hora de afrontar el desarrollo con esta nueva versión (Figura 3.3):

1. Tratar de hacer lo correcto por defecto.

2. Si no se puede conseguir que haga lo correcto, lanzar una excepción.

3. En la medida de lo posible, guiar a los usuarios hacia patrones de uso que presenten un riesgo significativo de corromper los datos en codificaciones no compatibles con ASCII.

Figura 3.3
La versión actual del lenguaje es Python 3 (desde 2008).

Aunque analizaremos las principales diferencias entre las versiones de Python 2 y Python 3 más adelante, se puede apuntar que los principales cambios introducidos en Python 3 son:

- Mejora del soporte Unicode.
- Separación entre cadenas Unicode y datos binarios.
- Ajuste de las funciones "print()" y "exec()" para ser más legibles y coherentes, siguiendo la filosofía de Python: ser un lenguaje limpio y legible.
- Cambios en la sintaxis.
- Incremento en el número de tipos de datos.
- Incremento en los comparadores.

3.4. Soporte y migración de Python 2

Como ya se ha dicho, el 1 de enero de 2020 finalizó oficialmente el soporte para Python 2; concretamente, para su versión más longeva y distribuida de todas, la 2.7. De esa manera, la Python Software Foundation (PSF), con Guido van Rossum a la cabeza, anunció que Python 2 ya no iba a recibir ninguna actualización de seguridad, soporte ante bugs o correcciones de versión.

La intención inicial de la PSF era dar por terminado el soporte a Python 2 con la versión 2.6, pero en 2010 se publicó la versión 2.7, incorporando a dicha versión parte de las novedades que debían aplicarse en la versión inicial de Python 3. Además, el período de mantenimiento de Python 2.7 se tuvo que duplicar, pasando de los habituales cinco años a diez, hasta 2020. Esto fue debido a la gran popularidad de la versión 2.7 de Python y a que era la versión por defecto en muchas distribuciones de Linux; por estas razones, la PSF dio por buena la idea de soportar la tarea de dar cobertura tanto a la rama actual en aquel momento, la 2.7, como a la rama en desarrollo que daría como resultado Python 3.

Por ello, desde la fundación ya se ha lanzado varias veces el mensaje de que, incluso si se detectase un grave fallo de seguridad o un problema con alguna biblioteca de Python 2, los voluntarios de la PSF no dedicarán ni un minuto a solventar o atender dichas incidencias, ya

Ejercicios resueltos

Practica el **ejercicio 3** con los conocimientos adquiridos hasta esta unidad.

que lo prioritario en estos momentos es el desarrollo y el despliegue de Python 3.

Más allá de eso, se ha recomendado a los rezagados que migren cuanto antes a la nueva versión su código en Python 2; para ello, se ha lanzado una guía de consejos para portar el código, así como una herramienta llamada 2to3, que es un compendio de funciones y métodos que identifican las estructuras de código en Python 2 y traducen este a Python 3.

Uno de los directivos de la PSF, Nick Coghlan, declaró que, gracias a los esfuerzos de toda la comunidad Python (no solo a los voluntarios de la fundación), la versión de Python 3 ya está lista para afrontar cualquier tarea que pudiera haber estado desarrollada en Python 2.

En cualquier caso, este cambio de versión no significa que no se pueda o que no se vaya a programar en Python 2; simplemente, esa versión ya no va a recibir más soporte ni actualizaciones. No sería de extrañar que alguna empresa tuviese *software* desarrollado y funcionando en Python 2; sin embargo, si bien no es imposible desarrollar en esa versión, ciertamente no es lo recomendable por lo expuesto que queda el código a brechas de seguridad y a una más que posible desactualización progresiva de funcionalidades conforme avance la tecnología y se tengan que incorporar nuevas librerías (Figura 3.4).

Figura 3.4
La Python Software Foundation recomienda migrar todo el código a Python 3.

3.5. Principales diferencias entre Python 2 y Python 3

El cambio de Python 2 a Python 3 no solo es sintáctico sino que, como veremos a continuación, se trata de un cambio de concepto. Las principales diferencias entre estas dos versiones son las siguientes:

3.5.1. Función “print”

Esta es quizá la diferencia más llamativa. En Python 3, “print” ha pasado a ser una función, así que la expresión que hay que sacar por pantalla debe ir encerrada entre paréntesis. En Python 2 no eran necesarios los paréntesis (en este aspecto, se parece mucho más a la sintaxis de Ruby).

```
#Python2
print ‘Bienvenidos al curso de Python’

#Python3
print (‘Bienvenidos al curso de Python’)
```

3.5.2. Operaciones matemáticas

En Python 2, si realizamos una división de números enteros, obtendremos siempre un resultado entero redondeado al más próximo (resultado de la división **1 / 4 0,25 → 0**). Si queremos obtener los decimales, bastará con incluir un decimal en el numerador o el denominador.

Por el contrario, en Python 3, siempre que ejecutemos una división como la anterior, obtendremos los decimales en el resultado; y si queremos forzar que el resultado solo sea un entero, será tan sencillo como incluir una barra “/” en la división (resultado de la división **1 // 4 0,25 → 0**); aunque no sea necesaria esta funcionalidad, también la incorpora Python 2.

```
#Python3
a = ‘Hola’
type(a)

<class ‘str’>

b = b’Mundo’
type(b)

<class ‘bytes’>

a+b
TypeError
Traceback (most recent call last)
<ipython-input-8-928fa1475685> in <module>()
      6 type(b)
      7
----> 8 a+b

TypeError: can only concatenate str (not “bytes”) to str
```

3.5.3. Cadenas de texto Unicode

En Python 2, las cadenas de texto del tipo "string" estaban codificadas en formato ASCII, así que cada carácter necesitaba 7 bits de información para ser almacenado. Esa característica no presentaba ningún problema cuando el idioma de entrada era el inglés; pero en el caso de que hubiera que usar algún carácter especial, acentos o la letra *Ñ* (que no está en ASCII), se tenía que especificar una codificación que lo soportase (como, por ejemplo, UTF-8) y guardar el archivo fuente con esa codificación, ya que daba un error en cualquier otro caso. Se tenía que especificar la codificación correcta para obtener una salida sin errores.

En Python 3, esto ha cambiado y ahora todas las cadenas de texto son Unicode (8 bits). Pueden ser almacenadas como texto (variables del tipo "str") o como datos binarios (bytes), pero es imposible la combinación de ambos tipos, ya que nos arrojaría una excepción "TypeError".

```
#Python3
a = 'Hola'
type(a)

<class 'str'>

b = b'Mundo'
type(b)

<class 'bytes'>

a+b
TypeError
Traceback (most recent call last)
<ipython-input-8-928fa1475685> in <module>()
      6 type(b)
      7
----> 8 a+b

TypeError: can only concatenate str (not "bytes") to str
```

3.5.4. Función "input"

Esta función se usa para recoger la entrada por teclado por parte del usuario. En Python 2 no se hacía ninguna conversión de los datos y se recogía tal cual, es decir: si lo que el usuario iba a introducir era un entero, la entrada debía ser de tipo "int"; pero si, por ejemplo, queríamos recoger una cadena de texto, había que usar la función que convierte los datos de entrada a tipo "str", que era la función "raw_input()".

```
#Python2
entrada = input ('Introduce un número: ')

Introduce un número : 1

print(type(entrada))

<class 'int'>

entrada = raw_input ('Introduce un número: ')

Introduce un número: 1

print(type(entrada))

<class 'str'>
```

Ejercicios resueltos

Practica el **ejercicio 4** con los conocimientos adquiridos hasta esta unidad.

Este tipo de especificación de la entrada ha desaparecido en Python 3, ya que la función "raw_input()" ya no existe y el tratamiento de la entrada lo toma de la función "input". Esto simplifica enormemente el tratamiento de los datos que entran por el teclado, ya que sabremos que serán de tipo "string" con antelación.

```
#Python3
entrada = input ('Introduce un número: ')

Introduce un número : 1

print(type(entrada))

<class 'str'>

entrada = raw_input ('Introduce un número: ')

Intorduce un número: 1
<class 'str'>
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-218a6fb2db9f> in <module>()
      9
     10
---> 11 entrada = raw_input ('Introduce un número: ')
     12
     13 I

NameError: name 'raw_input' is not defined
```

3.5.5. El método “next()”

De la misma manera que “print” ha pasado a ser una función en Python 3, también se ha simplificado el método “next()” de Python 2, ya que, si bien antes podía usarse como método o como función, en Python 3 solo existe como función. Se usa para recoger el siguiente elemento de una iteración.

```
#Python 2

bucle = (letra for letra in ‘python’)
print(next(bucle))

p

print(bucle.next())

y
```

Si intentamos utilizar “next” como un método en Python 3, se nos generará una excepción “AttributeError”.

```
#Python 3

bucle = (letra for letra in ‘python’)
print(next(bucle))

print(bucle.next())

p
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-11-6638c3b9365f> in <module>()
      6
      7
----> 8 print(bucle.next())
      9
     10

AttributeError: ‘generator’ object has no attribute ‘next’
```

3.5.6. Comparación de tipos

En Python 3, la comparación de tipos diferentes ya no devuelve un "true" o un "false", sino que arroja un error. Si comparamos una lista con una cadena de texto, por ejemplo:

```
from platform import python_version
print('Python' + python_version())

#sabremos en que versión de Python se ha ejecutado
```

En Python 2:

```
#Python2

print("Comparamos [1,2] > 'string' = "), [1,2] > 'string'

Comparamos [1,2] > 'string' = False

print("Comparamos (1,2) > 'string' = "), (1,2) > 'string'

Comparamos (1,2) > 'string' = True

print("Comparamos [1,2] > (1,2) = "), [1,2] > (1,2))

Comparamos [1,2] > (1,2)) = False
```

En Python 3:

```
#Python3

print("Comparamos [1,2] > 'string' = ", [1,2] > 'string')

Comparamos [1,2] > 'string' =
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-bd66c912913b> in <module>()
      1 #Python2
      2
----> 3 print("Comparamos [1,2] > 'string' = "), [1,2] > 'string'
      4
```

```
      5 #print("Comparamos (1,2) > 'string' = "), (1,2) > 'string'

TypeError: '>' not supported between instances of 'list' and 'str'


print("Comparamos (1,2) > 'string' = ", (1,2) > 'string')

Comparamos (1,2) > 'string' =
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-57de79b7f390> in <module>()
      3 #print("Comparamos [1,2] > 'string' = "), [1,2] > 'string'
      4
----> 5 print("Comparamos (1,2) > 'string' = "), (1,2) > 'string'
      6
      7 #print("Comparamos [1,2] > (1,2) = "), [1,2] > (1,2))

TypeError: '>' not supported between instances of 'tuple' and 'str'

print("Comparamos [1,2] > (1,2) = ", [1,2] > (1,2))

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-b22cad7ea7bb> in <module>()
      5 #print("Comparamos (1,2) > 'string' = "), (1,2) > 'string'
      6
----> 7 print("Comparamos [1,2] > (1,2) = ", [1,2] > (1,2))
      8
      9

TypeError: '>' not supported between instances of 'list' and 'tuple'

 print ('Comparamos 35 > 20 =', 35>20)

Comparamos 35 > 20 = True
```

3.5.7. Manejo de excepciones

Otro ejemplo en el que se reduce la sintaxis a una función es el manejo de excepciones, ya que en Python 2 se aceptaba su escritura con paréntesis o sin ellos, mientras que en Python 3 solo se permite declararlas con paréntesis. El resultado en Python 2 sería el siguiente:

```
#Python2
raise IOError, "error interno"
Traceback (most recent call last):
  File "Main.py", line 1, in <module>
    raise IOError, "error interno"
IOError: error interno
```

En Python 3 nos arrojaría un error de sintaxis:

```
#Python3
raise IOError, "error interno"

File "<ipython-input-19-d171998710b2>", line 2
raise IOError, "error interno"
  ^
SyntaxError: invalid syntax
```

3.5.8. Funciones "xrange" y "range"

Las funciones "xrange" y "range" nos permiten crear listas de números enteros que podemos reutilizar para las iteraciones de los bucles "for". La diferencia entre ambas estriba en el modo en el que se genera la lista, ya que esto implica diferencia de memoria utilizada y tiempo necesario para poder recorrer todos los elementos de la lista. En cualquier caso, la función "xrange" ya no está disponible en Python 3; y si se usa, obtendremos una excepción del tipo "NameError".

```
#Python2
#number-> ejecuciones de la instrucción

import timeit
print 'tiempo empleado',timeit.timeit('(str(n) for n in range(100))', number=100000)

Tiempo empleado 0.291787147522

print 'tiempo empleado',timeit.timeit('(str(n) for n in xrange(100))', number=100000)

Tiempo empleado 0.132493972778
```

Mientras que en Python 3:

```
#Python3
#number-> ejecuciones de la instrucción

import timeit
print ('tiempo empleado',timeit.timeit('(str(n) for n in range(100))', number=100000))

tiempo empleado 0.06071018100010406

print ('tiempo empleado',timeit.timeit('(str(n) for n in xrange(100))', number=100000))

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-21-88e5412524ec> in <module>()
      8
----> 9 print ('tiempo empleado',timeit.timeit('(str(n) for n in xrange(100))', number=100000))
     10

/usr/lib/python3.7/timeit.py in timeit(stmt, setup, timer, number, globals)
    231            number=default_number, globals=None):
    232     """Convenience function to create Timer object and call timeit method."""
--> 233     return Timer(stmt, setup, timer, globals).timeit(number)
    234
    235 def repeat(stmt="pass", setup="pass", timer=default_timer,

/usr/lib/python3.7/timeit.py in timeit(self, number)
    175         gc.disable()
    176         try:
--> 177             timing = self.inner(it, self.timer)
    178         finally:
    179             if gcold:

/usr/lib/python3.7/timeit.py in inner(_it, _timer)

NameError: name 'xrange' is not defined
```

3.5.9. Iterar un diccionario

En Python 2, era posible iterar los elementos con forma key-value en un diccionario mediante el método "iteritems()" o de "items()".

```
#Python2
#number -> ejecuciones de la instrucción

d = {'animal':'perro','vehículo':'coche'}
for x,y in d.iteritems():
    print x,'-->',y

vehículo --> coche
animal --> perro
```

En Python 3, desaparece "iteritems()" por razones de rendimiento (de la misma manera que "xrange") y, en caso de utilizarla, obtendremos una excepción del tipo "AttributeError", por lo que solo se podrá realizar a través de la función "items()".

```
#Python3
#number-> ejecuciones de la instrucción

d = {'animal':'perro','vehículo':'coche'}
for x,y in d.iteritems():
  print(x,'-->',y)

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-23-dd582ce6f3db> in <module>()
      3
      4 d = {'animal':'perro','vehículo':'coche'}
----> 5 for x,y in d.iteritems():
      6   print(x,'-->',y)
      7

AttributeError: 'dict' object has no attribute 'iteritems'


#Python3
#number -> ejecuciones de la instrucción

d = {'animal':'perro','vehículo':'coche'}
for e in d.items():
    print(e)

('animal', 'perro')
('vehículo', 'coche')
```

Resumen

- La Python Software Foundation (PSF) ha descontinuado el soporte de Python 2 para centrarse única y exclusivamente en el desarrollo de Python 3, que es la versión que actualmente está activa.

- El paso de Python 2 a Python 3 implica un cambio de concepto en muchos métodos y funciones (no es solo un cambio de versión en el que se incorporan mejoras), ya que la forma de escribir las órdenes y los resultados de algunas operaciones matemáticas han cambiado debido a la redefinición de estos.

- La PSF ha puesto una herramienta a disposición de los desarrolladores para migrar el *software* hecho en Python 2 a Python 3.

- Si bien se puede seguir desarrollando código en Python 2, no es nada recomendable, puesto que no se contará con, por ejemplo, actualizaciones de seguridad, lo que podría afectar a la integridad del código desarrollado.

4. Introducción al control de flujo y entradas/salidas de datos

Como ya se ha explicado anteriormente, un programa Python es un fichero con código fuente, cuya codificación suele ser UTF-8, que contiene sintaxis, órdenes y expresiones en Python.

El lenguaje está formado por elementos (*tokens*) que podrían agruparse en varios conjuntos o segmentos, entre los cuales podríamos destacar los siguientes (aunque no son los únicos):

- Palabras reservadas (*keywords*)
- Funciones integradas (*built-in functions*)
- Literales
- Operadores
- Delimitadores
- Identificadores

Para que el código fuente escrito en Python se ejecute, este ha de ser sintácticamente correcto, y para ello hemos de seguir una serie de pautas.

4.1. Palabras reservadas

Existe una serie de palabras propias del lenguaje Python, pero de uso interno, que no podemos usar como identificadores, funciones o variables; de hacerlo, se produciría un error de sintaxis, ya que estas palabras están bloqueadas para el usuario.

false	await	else	import	pass	none	break
except	in	raise	true	class	finally	is
return	and	continue	for	lambda	try	as
def	from	nonlocal	while	assert	del	global
not	with	async	elif	if	or	yield

Certificación

Es importante interiorizar el control de flujo y entrada/salida de datos, pues en numerosas preguntas de la certificación se hace referencia a este tema.

4.2. Líneas y espacios

Todo código de Python está dividido en líneas. Estas líneas pueden contener los caracteres necesarios para ejecutar una instrucción, declarar una variable, abrir un bloque condicional, etc.

Si en el código hay una línea en blanco, el intérprete de Python la ignorará. Para tener un código claro y visualmente cómodo para identificar los bucles, las jerarquías, la declaración de variables, etc., es necesario aplicar unas reglas de espaciado o de sangrado.

Para unir dos o más líneas, podemos emplear la barra inclinada hacia la izquierda "/".

Por otra parte, también existe la posibilidad de que queramos realizar varias acciones en una sola línea; para eso podemos concatenar tantas instancias como queramos usando como separador o delimitador de cada orden el punto y coma ";".

4.3. Comentarios y citas

En Python, la inclusión de comentarios en el texto se consigue con el carácter especial "#". Los comentarios son anotaciones que el intérprete de Python ignorará, pero que nosotros, los programadores, podemos (y debemos) leer, ya que nos aportan información sobre el código que hay escrito. Pueden ser aclaraciones, delimitaciones, referencias a otras funciones o cualquier tipo de información que nos ayude a interpretar y comprender el código.

Esto es especialmente importante cuando varios programadores emplean el mismo fichero de código fuente y se dejan anotaciones aclaratorias en el código que escribe cada uno para que otros programadores que revisen el fichero y sus instrucciones sepan cómo interpretarlo.

Los comentarios se pueden incluir en una línea separada del código o en la misma línea de la orden. También se pueden insertar comentarios que ocupen varias líneas.

```
# Esto es un comentario de una sola línea
var = 42

"""Esto es un comentario
```

```
que ocupa varias lineas"""
var = 42

var = 42 # Esto es un comentario en la misma línea de la operación
```

Los "strings" o cadenas de texto se delimitan con comillas en Python. Podemos usar comillas simples (' '), comillas dobles (" ") o incluso comillas triples (""" """). Cada una de ellas tendrá una funcionalidad, pero lo realmente importante es que, independientemente de si usamos las simples, las dobles o las triples, las que usemos al empezar a delimitar el "string" deben ser las mismas que al acabar; en otras palabras, el tipo de comillas de apertura debe ser igual que el de las de cierre.

Cada una de ellas tiene una serie de ventajas; por eso, elegiremos unas u otras según lo que queramos hacer en la declaración de "strings".

```
print('Mi estilo de música favorita es el Rock"')
Mi estilo de música favorita es el Rock"
```

Apreciamos aquí que hemos podido combinar ambas, pero las de cierre son iguales a las de apertura.

Complicando un poco más el ejemplo, supongamos que necesitamos sacar por pantalla las comillas dobles, como en el ejemplo anterior, pero también que aparezca alguna comilla simple. La solución nos la da lo que se denomina carácter de escape, que en Python se consigue con el carácter barra inversa o *backslash* ("\") antes de las comillas que queremos que Python no interprete como comillas de cierre y, de esta manera, se muestren.

```
print('Me gusta el rock\'n\'roll')
Me gusta el rock'n'roll
```

Puede darse un problema con las cadenas largas de caracteres, y es que deben estar en la misma línea de código, ya que, si hacemos un retorno de carro (Enter), Python interpretará como una orden la nueva línea de código. Si, principalmente por cuestiones de comodidad, fuese necesario partir el código continuando en otra línea pero deba formar parte de la instrucción original, se puede hacer acabando la línea con el carácter de escape ("\") y pulsando Enter a continuación.

```
print('El estilo de música que más me gusta \
es el "Rock\'n\'roll"')
El estilo de música que más me gusta es el "Rock'n'roll"
```

De la misma manera, tenemos caracteres de escape de nueva línea ("\n") y de tabulación ("\t"), para dar un pequeño formateo a una cadena de caracteres larga.

```
print('Los grupos de Rock suelen tener:\n-Batería\n-Bajo\n-Guitarra solita\n-Guitarra
rítmica\n-Voz')

Los grupos de Rock suelen tener:
-Batería
-Bajo
-Guitarra solita
-Guitarra rítmica
-Voz

print('Mis grupos de "Rock" favoritos son \n-Led Zeppelin\t-Dire Straits\t-Aerosmith')

Mis grupos de "Rock" favoritos son
-Led Zeppelin  -Dire Straits  -Aerosmith
```

Todos los casos anteriores pueden darse de formas más o menos complejas, por lo que la escritura con caracteres de escape podría complicarse de manera que ni la escritura del código Python fuese fluida ni la interpretación del código fuera visualmente demasiado directa.

Por esta razón entran en juego las triples comillas, que nos permiten insertar una cadena de texto con todas las líneas que queramos, y que, internamente, podamos poner todas las comillas dobles o simples que necesitemos sin necesidad de caracteres de escape, ya que el formato dentro de las comillas triples se respetará.

```
print("""" Mi estilo de música favorita es el "Rock'n'Roll
Mis grupos favoritos son:
Nirvana   Foo Fighters    Muse
Un grupo de rock suele tener
- Batería
- Bajo
- Guitarra solista
- Guitarra rítmica
- Voz""""")
```

```
Mi estilo de música favorita es el “Rock’n’Roll
Mis grupos favoritos son:
Nirvana   Foo Fighters    Muse
Un grupo de rock suele tener
- Batería
- Bajo
- Guitarra solista
- Guitarra rítmica
- Voz
```

4.4. Operadores

Los operadores son caracteres especiales que se usan en operaciones lógicas y aritméticas. Pueden ser usados dentro de bucles o en líneas de código independientes.

+	-	*	**	/	//	%	@
<<	>>	&	|	^	~		
<	>	<=	>=	==	!=		

4.5. Delimitadores

Los caracteres denominados delimitadores tienen la función de acotar o limitar expresiones.

‘	“	#	\			
(	)	[	]	{	}	
,	:	.	;	@	=	->
+=	=	*=	/=	//=	%=	@=
&=	|=	^=	>>=	<<=	**=	

Certificación

Es importante interiorizar los operadores, delimitadores e identificadores, pues en numerosas preguntas de la certificación se hace referencia a estos.

4.6. Identificadores

Los identificadores sirven para declarar una variable, módulo, clase, función u objeto. Han de seguir ciertas reglas y hemos de tener en cuenta que:

- Python diferencia entre mayúsculas y minúsculas.
- No se admiten caracteres de puntuación como @, $ o %.
- Los nombres de las clases deben empezar por una letra mayúscula.
- El resto de los identificadores que son de clase empiezan por letra minúscula.
- Si queremos declarar un identificador privado, deberá empezar por guion bajo.
- Si el identificador acaba con dos guiones bajos, estaremos ante un identificador especial definido por Python.
- El primer carácter del identificador debe ser una letra.
- Es recomendable utilizar caracteres ASCII para estandarizar la declaración de identificadores y que el código pueda ser entendido fácilmente por usuarios de otros países que puedan tener un juego de caracteres diferentes.

4.7. Funciones integradas y adicionales

Una función es un compendio de órdenes bien estructurado que permite ejecutar una serie de instrucciones. El hecho de que las instrucciones estén encapsuladas o acotadas dentro de una función favorece la modularización y la reutilización de estas.

Python, de la misma forma que tiene palabras reservadas que no se pueden utilizar para declarar funciones o variables, también tiene incorporadas por defecto una serie de funciones; esto significa que, independientemente de las funciones que nosotros creemos, estas van a estar siempre disponibles.

Podemos redefinir estas funciones por código, pero no podemos crear funciones con el mismo nombre, ni que coincidan con las palabras reservadas.

A la hora de escribir un programa, además de estas funciones integradas, podemos echar mano de nuevas funciones que se encuentran definidas en otros ficheros, llamados **módulos**. Python incluye una biblio-

teca de módulos (llamada **biblioteca estándar**), cada uno de los cuales está especializado en un tipo de tarea concreta: diseño de gráficas, cálculos matemáticos avanzados, inteligencia artificial, etc.

Además, gracias a la gran comunidad de programadores de Python, hay disponibles infinidad de módulos desarrollados por programadores. Se pueden consultar en el Python Package Index, más conocido como PyPI, sus siglas en inglés.

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	slice()
ascii()	enumerate()	input()	oct()	statimethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	_ _ import _ _()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

4.8. Estructuras de control de flujo condicionales

Una estructura de control condicional nos permite modificar el flujo de la información dentro del programa a partir de una o varias condiciones. Esto se hace con una serie de símbolos a modo de comparadores o evaluadores de una condición, y podrá dar como resultado dos posibles caminos: verdadero o falso. A estos símbolos se les llama **operadores de comparación** (o **relacionales**).

Símbolo	Significado	Ejemplo	Evaluación
==	Igual que	42 == 88	False
!=	Distinto de	42 != 88	True
<	Menor que	1 < 2	True
>	Mayor que	12 > 120	False

Símbolo	Significado	Ejemplo	Evaluación
<=	Menor o igual que	42 <= 42	True
>=	Mayor o igual que	40 >= 52	False

Para poder evaluar más de una condición a la vez, se utilizan los denominados operadores lógicos.

Operador	Ejemplo	Resultado	Evaluación
and	7 == 9 and 7 < 15	False and True	False
and	90 < 120 and 3 > 1	True and True	True
and	15 < 20 and 15 > 13	True and False	False
or	42 == 42 or 12 < 37	True or False	True
or	37 > 35 or 15 < 22	True or True	True
xor	40 == 40 xor 6 > 2	True o True	False
xor	40 == 40 xor 6 < 2	True o False	True

Para tener una visión clara del orden en el que se ejecutarían las comparaciones y las evaluaciones con estos comparadores, debemos tener en cuenta la siguiente tabla de prevalencia:

Operador	Definición
+n, -n	Operadores aritméticos unitarios para números positivos o negativos
**	Operador aritmético de potencia/exponente
NOT	Operador lógico de negación
*, /, //, %	Operadores aritméticos de multiplicación, división, división entera y resto
+, -	Operadores aritméticos de suma y resta
<, <=, =>, >	Comparadores lógicos de magnitud
==, !=	Comparadores lógicos de igualdad o desigualdad
AND	Operador lógico Y
OR	Operador lógico O
=, +=, -=, *=, /=, //=, %=	Operadores de asignación, incremento y decremento

Además de con operadores, las condiciones se evalúan con estructuras de flujo condicionales. Estas tres estructuras, que vienen definidas por tres palabras clave, se verán más en detalle en el módulo 4, pero conviene conocer cuál es el flujo que sigue la información según la condición con un ejemplo muy simple.

- **IF:** comparación que se hace y que, en caso de evaluarse como verdadera, hará que se ejecute el código inmediatamente inferior.

```
var1 = 1
var2 = 2
var3 = 3
var4 = 4

if(var1 < var2):
  print('La variable 1 es menor que la variable 2')

La variable 1 es menor que la variable 2
```

- **ELIF:** añade a la evaluación anterior otra condición. Es como añadir un operador AND a una comparación IF.

```
var1 = 1
var2 = 2
var3 = 3
var4 = 4

var1 = (var2 + var3)

if var1 == 4: #Esta condición evalua si var1 es exatamente igual a 4
  print ('var1 es igual a 4')
elif var1 == 5:
  print ('var1 es igual a 5')
elif var1 == 6:
  print ('var1 es igual a 6')
else:
  print ('No se cumple ninguna condición')

var1 es igual a 5
```

- **ELSE:** código que se ejecuta en el supuesto de que la evaluación del IF no haya sido evaluada como verdadera.

```
var1 = 1
var2 = 2
var3 = 3
var4 = 4

if (var1 > var2):
  print('La variable 1 es mayora que la variable 2')
else:
  print('La variable 1 es menor que la variable 2')

La variable 1 es menor que la variable 2
```

4.9. Estructuras de control de flujo iterativas

Existe otro elemento para controlar el flujo de la información en nuestros programas: las estructuras de control de flujo iterativas, que, a diferencia de las condicionales, nos permiten ejecutar una misma serie de instrucciones de manera repetida mientras se esté cumpliendo una condición en concreto.

De igual forma que en las anteriores estructuras (las condicionales), las estructuras iterativas se verán más en detalle en el módulo 4, pero las vemos con un pequeño ejemplo para evaluar el funcionamiento de los distintos métodos iterativos.

En Python existen dos estructuras iterativas:

- **WHILE:** ejecuta la misma acción "mientras" se esté cumpliendo una condición.

```
year = 2000
while (year<2012):
  print (year)
  year +=1

2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
```

- **FOR:** permite iterar o repetir una misma acción un número de veces determinado. Puede actuar sobre variables complejas, del tipo tupla o lista.

```
gruposRock_VLC = ['Revolver', 'El pez volador', 'Los Zigarros',
'Seguridad social']

for nombre in gruposRock_VLC:
```

```
  print (nombre)

Revolver
El pez volador
Los Zigarros
Seguridad social
```

Resumen

- Para que un programa escrito en Python sea correcto, ha de respetar una serie de reglas de sintaxis y de escritura de ciertos elementos esenciales del lenguaje.

- Se han de respetar las palabras reservadas del lenguaje; no le podemos poner a una de nuestras variables un nombre que coincida con el de una palabra reservada, ya que nos dará un error y nuestro código no se ejecutará.

- Debemos conocer todos los operadores del lenguaje Python, tanto los lógicos como los aritméticos, ya que son la base de cualquier cálculo o comparación dentro del código.

- Existen ciertos bucles, tanto iterativos como decisorios, que nos permiten variar el flujo de la información de nuestro programa a partir de una serie de variables y condiciones.

5. Errores frecuentes y depuración de código

El hecho de conocer otros lenguajes de programación o haber programado o diseñado *software* en pseudocódigo con cierta asiduidad da una ventaja competitiva al programador de Python, pues, como ya se ha visto, hay conceptos comunes a muchos lenguajes (como pueden ser los bloques iterativos o decisorios); sin embargo, esto puede suponer también un leve hándicap a la hora de programar en Python, ya que se pueden heredar malos hábitos de un lenguaje a otro, o tener la costumbre de escribir delimitadores que nos parecen comunes pero que en Python no lo son, lo que nos llevará a errores.

Vamos a ver cómo mejorar de entrada esas costumbres y corregir esos vicios, ya que en la mayoría de los casos la codificación se simplifica bastante en este lenguaje.

Para saber más

En Java y PHP, las líneas de código se dan por finalizadas con el delimitador punto y coma ";", mientras que en Python no es necesario, y, si usamos ese delimitador, se producirá un error.

5.1. Mala indentación

A la hora de escribir funciones, bucles o bloques de código, hemos de tener en cuenta la sangría que debe llevar cada línea. Esto nos ayudará a que el código sea más fácilmente identificable, más visual; si no lo hacemos, Python nos lanzará un error, ya que nos fuerza a indentar cada línea para respetar la jerarquía del código. Se asume que todas las líneas que se encuentran tabuladas a la misma distancia (con la misma sangría) están dentro del mismo bloque de código.

```
a = 1
b = 2
c = 3
d = 4

if (a<d):
print('A es menor que B')

File "<ipython-input-40-4ba3242fed4a>", line 7
print('A es menor que B')
    ^
IndentationError: expected an indented block
```

Certificación

Es importante interiorizar los errores frecuentes y depuración de código, pues en numerosas preguntas de la certificación se hace referencia a este tema.

```
a = 1
b = 2
c = 3
d = 4

if (a<d):
  print('A es menor que B')

A es menor que B
```

5.2. Llamada errónea de variables

Python distingue entre mayúsculas y minúsculas a la hora identificar las variables. Es lo que técnicamente se conoce como un lenguaje *case-sensitive*. Por ello, hemos de asegurarnos de que llamamos la misma variable que hemos declarado.

```
variable1 = "Gato"
variable2 = "Perro"

print ('Mi animal favorito es el ', Variable2)
------------------------------------------------------------
NameError
Traceback (most recent call last)
<ipython-input-42-0842fcaa327b> in <module>()
      2 variable2 = "Perro"
      3
----> 4 print ('Mi animal favorito es el ', Variable2)

NameError: name 'Variable2' is not defined
```

Como podemos ver, Python nos dice que no reconoce la variable, ya que hemos intentado llamarla en minúscula, cuando, en realidad, "Variable_2" empieza con mayúscula.

```
variable1 = "Gato"
variable2 = "Perro"

print ('Mi animal favorito es el ', Variable1)

Mi animal favorito es el  Gato
```

5.3. Usar palabras reservadas como nombres de variables

Si usamos como identificador de una variable el nombre de una palabra reservada, no solo no podremos operar con esa variable, sino que además Python nos generará un error y el intérprete no podrá seguir su función. Hemos de seguir una disciplina lo más coherente posible a la hora de nombrar e identificar variables y nombres de funciones, ya que, además de no generarnos ningún error, hará que nuestro código sea más claro y fácil de interpretar visual y conceptualmente.

```
stop = "Parar"
go = "Avanzar"
refresh = "Recargar"
break = "Interrumpir"

File "<ipython-input-44-25ab433b1b1c>", line 4
break = "Interrumpir"
^
SyntaxError: invalid syntax
```

5.4. Los espacios como guion bajo

Es más que recomendable evitar generar ficheros o carpetas con espacios en el nombre. Eso dificulta muchas veces el acceso a rutas y puede que nos cause más de un problema a la hora de llamar a una biblioteca o abrir algún tipo de archivo, por lo que se recomienda renombrar sustituyendo los espacios en blanco por el carácter de guion bajo (_).

5.5. Usar operando de asignación en lugar de igualdad

Cuando hagamos una comparación entre dos variables, valores u objetos, esta se realizará mediante el operador de igualdad (==); es muy común equivocarse y, siguiendo el lenguaje natural, intentar hacer la comparación solo con el operador de asignación (=). Hay que recordar que este operador coloca un objeto o valor dentro de una variable y, obviamente, no ejecuta ni evalúa ninguna comparación.

```
a = 12
b = 24
c = 42
if (a=b):
  print ('Son iguales!')
```

```
File “<ipython-input-45-035025374887>”, line 4
if (a=b)
^
SyntaxError: invalid syntax

a = 12
b = 24
c = 42
if (a==b):
  print (‘Son iguales!’)
else:
  print (‘Son distintas!’)
```

5.6. No finalizar correctamente una línea

A diferencia de PHP o Java, no es necesario un carácter de finalización de línea como puede ser el punto y coma (;), pero a veces sí es necesario acabar una declaración de función o bucle con los dos puntos (:); concretamente, al final de las sentencias de tipo:

- Try
- If
- Elif
- Else
- For
- While
- Class
- Def

```
a = “Nirvana”
b = “KISS”

if (a== b)
  print (a, “ es igual a “, b)
else
  print (a, “ no tiene nada que ver con “, b)
```

```
File “<ipython-input-49-89e13189efa3>”, line 4
if (a == b)
    ^
SyntaxError: invalid syntax
a = “Nirvana”
b = “KISS”

if (a== b):
  print (a, “ es igual a “, b)
else:
  print (a, “ no tiene nada que ver con “, b)

Nirvana  no tiene nada que ver con  KISS
```

Resumen

- Hay una serie de errores frecuentes a la hora de programar en Python. Si se conocen otros lenguajes de programación de alto nivel (como Java o PHP, por ejemplo), se suelen heredar costumbres relativas a la sintaxis de estos, que guarda parecido con la de Python, pero dan pie a incorrecciones.

- La indentación es muy importante en Python, ya que a partir de ella se evaluarán los bloques de código y las jerarquías dentro del mismo; por esa razón, si no escribimos con las sangrías correctas, nuestro código nos dará error y no llegará a ejecutarse.

- Otro elemento importante es que Python es case-sensitive, es decir, diferencia entre mayúsculas y minúsculas. Esto provoca muchos errores a la hora de llamar variables, y, si bien es fácil de corregir, es un detalle que nos puede hacer perder mucho tiempo

Índice