Análisis y minería de textos con PYTHON

Rafael Caballero Roldán
Enrique Martín Martín
Adrián Riesco Rodríguez
Universidad Complutense de Madrid

ÍNDICE

PRÓLOGO .... **IX**

CAPÍTULO 1. EL LENGUAJE DE PROGRAMACIÓN PYTHON **1**

Introducción 1

Programas y lenguajes 1

¿Por qué Python? 2

Cómo aprender Python 2

El entorno Jupyter Notebooks 3

Google Colaboratory 3

Subir ficheros de datos a Google Colaboratory 4

Instalación a través de Anaconda 5

¿Los Jupyter Notebooks son programas en Python? 6

Tipos simples en Python 6

Tipos numéricos 7

Tipo lógico 8

Variables 9

Secuencias 10

Concatenación 11

Acceso por posición 12

Otras operaciones para secuencias 13

Operaciones específicas para strings 14

Operaciones específicas para listas 18

Instrucciones condicionales 19

Iterando 21

Listas intensionales 24

Diccionarios 25

Funciones de usuario 27

Uso de bibliotecas Python 30

Importar bibliotecas 30

Instalar bibliotecas ... 32

CAPÍTULO 2. FICHEROS ... 35
Introducción ... 35
Ficheros de texto ... 35
Hojas de cálculo ... 39
CSV ... 49
JSON ... 55

CAPÍTULO 3. PALABRAS Y ELEMENTOS BÁSICOS ... 59
Introducción ... 59
Extracción de unidades léxicas con spaCy ... 60
Tipos de símbolos ... 61
Palabras vacías ... 65
Nubes de palabras ... 68
Modelos pre-entrenados ... 70
¿Qué es un modelo en spaCy? ... 70
Instalación y carga de los modelos ... 71
Los componentes del modelo ... 72
División de un texto en oraciones ... 73
Lematización ... 77
Extracción de lemas con spaCy ... 77
Aplicación: contando palabras ... 78
Extracción de raíces ... 80

CAPÍTULO 4. LOS ELEMENTOS DE LA ORACIÓN ... 83
Introducción ... 83
Análisis morfológico de palabras ... 84
Análisis sintáctico de oraciones ... 89
Detección de entidades con nombre ... 99
Otras bibliotecas para la detección de entidades con nombre ... 101
Referencias ... 102

CAPÍTULO 5. ANÁLISIS SEMÁNTICO ... 103
Introducción ... 103
Análisis de sentimiento ... 104
Semejanza entre palabras y entre frases ... 110
Modelado de temas ... 116
Análisis Semántico Latente ... 118
Asignación Latente de Dirichlet ... 121
Asignando tema a nuevos textos ... 124

©RC Libros

Mejorando el modelado de temas 126

CAPÍTULO 6. ANÁLISIS DE TEXTOS MEDIANTE MÉTRICAS 129
Introducción 129
Estadística descriptiva 132
Facilidad de lectura 134
Distancia de dependencia 137
Proporción de categorías gramaticales 140
Coherencia de textos 141
Visualización de los análisis 143
Creación de tablas en pandas 145
Diagrama de barras de la longitud de las frases 148
Diagrama de barras de varias medidas 150
Histograma de longitudes de frase 153
Diagrama de franjas sobre la distribución de longitudes de frase 156
Diagrama de caja de longitudes de frase 157
Diagrama de barras apiladas de proporciones de categorías gramaticales 159
Referencias 161

ÍNDICE ANALÍTICO 165

1 EL LENGUAJE DE PROGRAMACIÓN PYTHON

INTRODUCCIÓN

Comenzamos con una breve introducción al lenguaje de programación Python, así como al entorno de programación que vamos a utilizar. El nivel que se asume es totalmente inicial así que, aunque no se tenga experiencia alguna en programación, con un poco de esfuerzo aprenderemos lo suficiente para entender y poder aplicar los contenidos del resto del libro. Hay que señalar en todo caso que este capítulo no pretende ser una introducción exhaustiva al lenguaje, lo que requeriría (al menos) un libro completo. Afortunadamente, Python no nos interesa en sí mismo sino como herramienta para el tratamiento y procesamiento de textos y los conceptos básicos que necesitaremos en el resto del libro no nos obligan a profundizar excesivamente, tarea que dejamos para el lector que quiera seguir aprendiendo este lenguaje de programación mediante otros textos más orientados a su enseñanza.

Programas y lenguajes

Cuando programamos, nuestro objetivo es indicarle al ordenador qué debe hacer por medio de *programas*: conjuntos de instrucciones simples que le desglosarán nuestro objetivo paso a paso, de forma muy detallada y precisa. Cada programa debe estar escrito en un *lenguaje de programación* adecuado. Los más famosos para el tratamiento de datos son Python y R, pero hay muchos otros como Java, C++, etc. Cada uno tiene sus características, sus reglas gramaticales que, como veremos, debemos cumplir al detalle porque de lo contrario obtendremos un error que impedirá que el programa funcione.

Un detalle importante para comprender bien el papel de los lenguajes de programación: cada ordenador *entiende* realmente un solo lenguaje, el lenguaje máquina de su procesador. Como este lenguaje es muy complicado para una persona (decimos que es de *bajo nivel*), normalmente utilizaremos otros lenguajes de programación más cer-

canos a la forma de razonar humana para expresarnos, los llamados *lenguajes de alto nivel*. Un programa *traductor* o *intérprete* se encargará de convertir las instrucciones de este lenguaje de más alto nivel a lenguaje máquina. Para nosotros este proceso será transparente: nos parecerá que el ordenador está usando directamente el lenguaje de programación en cuestión, digamos Python, como si lo entendiera, pero en realidad estará utilizando siempre su lenguaje máquina gracias al trabajo del intérprete.

¿Por qué Python?

De entre todos los lenguajes de programación disponibles hemos elegido Python por ser el lenguaje más adecuado para el tratamiento de datos en general y de datos textuales en particular, y además está disponible de forma gratuita. Aunque fue creado en 1991, antes del *boom* de los datos, Python es un lenguaje versátil y ha sabido adaptarse perfectamente a este fenómeno. Podemos destacar tres razones principales.

En primer lugar, Python es un lenguaje fácil de aprender y utilizar, lo que lo convierte en una herramienta adecuada para quienes no tienen experiencia previa en programación. La sintaxis es clara y sencilla, sin la gran cantidad de «adornos» que precisan otros lenguajes más complejos. Con muy poco esfuerzo se puede pasar de no saber programar en absoluto a realizar programas que realicen tareas interesantes. Y esto lo lograremos gracias a la segunda razón, la enorme cantidad de *bibliotecas* disponibles. Podemos entender una biblioteca (también conocidas por la mala traducción *librerías* del inglés *library*) como nuevas funcionalidades que podemos incorporar a nuestro programa y que permiten realizar tareas complejas de forma sencilla. En nuestro caso una biblioteca puede permitirnos, por ejemplo, analizar sintácticamente un texto de forma automática, algo que no permite Python por defecto y que nos llevaría muchos días de trabajo si tuviéramos que hacerlo por nuestra cuenta. Además, y esta es nuestra tercera razón, en Python, especialmente si se utilizan entornos interactivos como *Jupyter Notebooks* o *Jupyter Lab*, podremos ir probando fragmentos pequeños de código según se escriben, sin tener que esperar a tener un programa completo, lo que nos permitirá aprender de forma incremental y evaluar rápidamente los resultados de diferentes alternativas.

Estas razones, entre otras, han hecho de Python uno de los lenguajes más populares, lo que a su vez hace que cada día haya más bibliotecas y más recursos en internet para aprender a sacarle el máximo partido.

Cómo aprender Python

El secreto para aprender cualquier lenguaje de programación se parece al de aprender un idioma nuevo: lo mejor es hablarlo, practicar. Más incluso en el caso de Python, donde no se requieren conceptos complejos para empezar y se pueden probar pequeños fragmentos de código desde el principio. Solo hace falta un poco de empeño y algo

©RC Libros

de tolerancia a la frustración: seguro que al principio obtendremos errores difíciles de entender, estaremos mirando el código durante media hora o incluso lo abandonaremos durante un tiempo dejándolo por imposible, solo para volver y descubrir finalmente que faltaba una coma, un paréntesis, que una variable estaba mal definida… paciencia. Aunque sirva de flaco consuelo, es necesario reconocer que a los que llevamos años programando nos sigue ocurriendo lo mismo, solo que lo sobrellevamos mejor porque ya hemos vivido la experiencia repetidas veces y porque sabemos que el resultado final suele compensar todos los desvelos.

Para empezar a programar lo primero que debemos hacer es disponer del lenguaje, veamos cómo hacerlo.

EL ENTORNO JUPYTER NOTEBOOKS

Aunque se puede programar en Python en muchos entornos diferentes, nosotros vamos a utilizar el conocido como *Jupyter Notebooks* (o en su versión en castellano *cuadernos Jupyter*), que nos permitirá probar el código de forma incremental tal y como indicamos en el apartado anterior. Veamos cómo instalarlo a través de dos herramientas gratuitas.

Google Colaboratory

La empresa *Google* nos ofrece un entorno gratuito en la nube para cargar y ejecutar Jupyter Notebooks. Es quizás la forma más sencilla de empezar ya que no requiere instalar nada; solo precisamos de un navegador web y una cuenta de *Google* para identificarnos. Nuestros *notebooks* o cuadernos se almacenarán en *Google Drive* aunque podremos descargarlos o subirlos desde nuestro ordenador local cuando deseemos. Podemos entrar en Google Colaboratory (*Colab* para los amigos, entre los que nos contamos desde ahora), a través de

`https://colab.research.google.com/`

En la primera pantalla podemos seleccionar «*Nuevo cuaderno*». En caso de que nos hayamos saltado esta primera pantalla siempre podemos seleccionar el menú «*Archivo + Nuevo cuaderno*». En ambos casos aparecerá nuestro primer cuaderno con un aspecto similar al que se ve en la figura 1.1.

La casilla o celda sombreada, que en la figura 1.1 muestra el mensaje «¡Hola Colab!», y que a nosotros nos aparecerá de momento vacía, es una *casilla de código*. No resistamos más la tentación y escribamos nuestra primera instrucción, siguiendo el ejemplo de la figura. Hay que ser cuidadoso de escribir todo exactamente igual, **`print`** en minúsculas, luego un paréntesis, luego el mensaje entre comillas dobles (las mismas para abrir que para cerrar, esas que en muchos teclados están sobre la tecla con el 2) y finalmente el paréntesis de cerrar. Y ya está: nuestro primer código en Python.

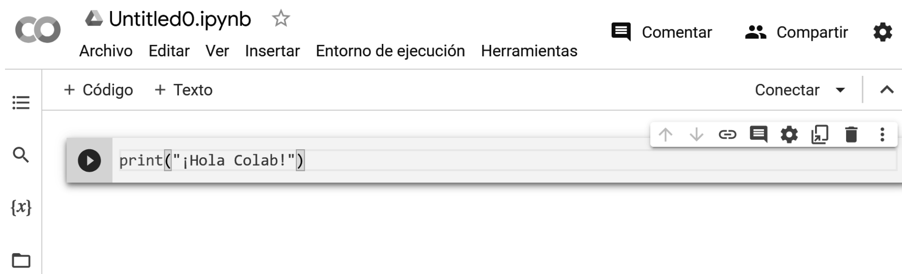

Figura 1.1: Un cuaderno en Google Colab

Para ejecutarlo solo debemos pulsar en el triángulo situado en la parte izquierda de la casilla de código. Si todo va bien, tras unos segundos justo debajo de la casilla aparecerá `¡Hola Colab!`, porque **`print`** sirve justo para esto, para escribir mensajes.

Además de código, los cuadernos nos permiten incluir casillas de texto, que corresponden con explicaciones que el ordenador ignorará y que podemos incluir para presentar o explicar mejor nuestro código. En cualquier momento podemos añadir una casilla de código pulsando en *+Código* y una de texto en *+Texto*. Si lo deseamos las casillas se pueden mover arriba y abajo en relación con otras casillas con las flechas que se ven a la derecha de cada una de ellas, o borrarlas pulsando sobre la papelera en esa misma zona. En el libro utilizaremos sobre todo casillas de código, pero recomendamos al lector acostumbrarse a documentar sus cuadernos con casillas de texto que expliquen el código.

También es una buena costumbre poner un nombre adecuado al cuaderno. Para ello haremos clic sobre *Untitled0.ipynb* y elegiremos un nombre como *hola.ipynb*, pero respetando la extensión *.ipynb* que corresponde a los Jupyter Notebooks. Así, la próxima vez que entremos en *Colab*, podremos volver a nuestro cuaderno con la opción *Archivo + Abrir cuaderno* (o *Abrir bloc de notas*, según la versión), que nos mostrará una lista con todos los cuadernos utilizados recientemente.

Subir ficheros de datos a Google Colaboratory

En nuestro caso, al trabajar con datos textuales, a menudo tendremos el texto a analizar en un fichero externo al que debe poder acceder de forma sencilla nuestro programa. Dado que en *Colab* estamos trabajando en la nube, debemos «subir» los ficheros de datos para que resulten accesibles.

Afortunadamente el proceso es muy sencillo: fijémonos de nuevo en la figura 1.1, pero esta vez prestando atención a la parte izquierda, que nos ofrece diversas funcionalidades. En nuestro caso estamos interesados en el símbolo que representa una carpeta de datos, justo debajo de la $\{x\}$. Si hacemos clic en este símbolo, se desplegará un panel en la izquierda con los ficheros de datos accesibles. En particular veremos por

©RC Libros

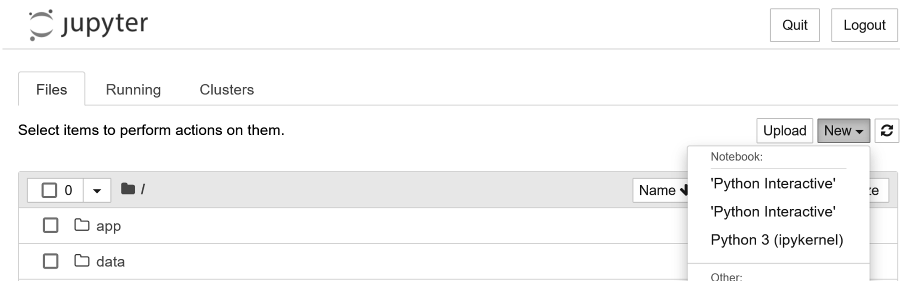

Figura 1.2: El navegador de Jupyter Notebook

ejemplo una subcarpeta con nombre *sample_data*. Para cargar aquí nuestro fichero de datos haremos clic con el botón derecho del ratón en el área indicada (por ejemplo, justo debajo de *sample_data*) y en el menú emergente elegiremos la opción *Subir*. Al hacerlo se nos abrirá una ventana que nos permitirá seleccionar directamente el archivo a subir a *Colab*. Otra forma de lograr lo mismo es arrastrar los ficheros a esta zona, pero por desgracia a veces «arrastrar y soltar» no es tan fácil como parece, por lo que nosotros preferimos la primera posibilidad.

Un aspecto negativo que debemos tener en cuenta: aunque nuestros cuadernos seguirán ahí cada vez que entremos en *Colab*, ya que se almacenan en nuestro *drive*, no sucede lo mismo con los ficheros de datos, que se eliminan al terminar la sesión por lo que deberemos subirlos cada vez. Por ello si queremos trabajar de forma habitual con ficheros sin la molestia de tener que subirlos una posibilidad puede ser utilizar el entorno Jupyter en nuestro propio ordenador instalándolo por ejemplo a partir de *Anaconda* como explica el siguiente apartado.

Instalación a través de Anaconda

Aunque la opción que nos propone *Google* mediante *Google Colab* es la más sencilla, también tiene sus limitaciones y puede que con el tiempo estemos interesados en realizar programas de tamaño mayor o simplemente no queramos depender de la conexión a internet y prefiramos tener Python y los Jupyter Notebooks disponibles en nuestro ordenador. Si es así, una posibilidad sencilla es instalar un producto como *Anaconda* que de forma gratuita nos proporciona múltiples entornos, entre otros los Jupyter Notebooks. Tras instalar *Anaconda* ejecutaremos el producto y de entre las opciones de las que dispone elegiremos *Jupyter Notebooks*. Al hacerlo y tras unos segundos, se nos abrirá en el navegador para que seleccionemos un cuaderno ya existente o creemos uno nuevo, como muestra la figura 1.2.

En el navegador podremos abrir un cuaderno ya existente moviéndonos por las carpetas, incorporar uno a la carpeta actual con *Upload* o crear uno nuevo con *New*. En nuestro caso pulsaremos en *New* y seleccionaremos *Python 3*. El cuaderno que se abrirá

será similar al de *Colab* que se muestra en la figura 1.1, con algunas pequeñas diferencias (puede que para ejecutar el código tengamos que dar a un botón *Run* en lugar de al triángulo de la izquierda como en *Colab*), pero en general el comportamiento será muy similar.

También se puede instalar la parte de Jupyter Notebooks sin *Anaconda*, ahorrándonos el paso intermedio, pero esto depende de cada sistema operativo por lo que deberemos buscar información específica en cada caso. En todo caso, y para evitar confusiones, en este texto hablaremos de *Anaconda* cuando se trate de los cuadernos instalados en nuestro ordenador, y de *Colab* para referirnos al entorno *Google Colaboratory*.

¿Los Jupyter Notebooks son programas en Python?

A estas alturas puede que el lector esté un poco confuso; empezamos hablando de Python pero de pronto hablamos sin parar de cuadernos. ¿Son realmente lo mismo? La respuesta es que los cuadernos contienen código en el lenguaje Python pero también más cosas, como las casillas de texto, imágenes o la misma posibilidad de ejecutar las casillas de código una a una, mientras que en un programa Python ejecutaríamos todas las casillas de código a la vez (cosa que, por cierto, también se puede hacer en el cuaderno «(*Entorno de Ejecución* + *Ejecutar Todo*» en *Colab*, y en *Anaconda* con «*Cell* + *Run All*»). Esta diferencia se aprecia en la *extensión* de los archivos, que será *.ipynb* en los cuadernos y *.py* en Python. Resumiendo: en los cuadernos estamos escribiendo en Python, ciertamente «adornado» con texto e imágenes además de con posibilidades de ejecución ampliadas, pero Python al fin y al cabo. Una forma habitual de proceder por los desarrolladores en Python es utilizar en primer lugar los cuadernos para ir probando el código poco a poco y al final, cuando ya se tiene un programa completo que vamos a ejecutar completo muchas veces, grabar el fichero como Python puro. Esto se hace en *Colab* con «*Archivo* + *Descargar* + *Descargar .py*», y en *Anaconda* en «*File* + *Download As* + *Python*».

TIPOS SIMPLES EN PYTHON

Los valores en Python, como en muchos otros lenguajes de programación, se agrupan en conjuntos coherentes llamados *tipos*. Aunque los tipos no se suelen ver explícitamente en Python es importante conocerlos porque cada tipo tiene su conjunto de operaciones predefinidas que podemos emplear para producir nuevos valores. Por ejemplo, podemos sumar dos números enteros (números sin parte decimal) de forma que diremos que la expresión `3+4` es correcta en Python (técnicamente se dice que está *correctamente tipada*), mientras que la expresión `3+"hola"` nos dará un error de tipo porque no es posible sumar un número y un mensaje, cuyo tipo en Python es *string* (en castellano se suele llamar a este tipo *cadena de caracteres*).

Comencemos por ver dos tipos básicos: los números y los valores lógicos.

©RC Libros

Tipos numéricos

Python incluye tres tipos de números: enteros, reales y números complejos. Solo nos vamos a interesar por los dos primeros porque los números complejos no se suelen utilizar en el procesamiento de textos.

Como hemos anticipado los números enteros son números sin parte decimal, y corresponden con el tipo de Python representado por la palabra **`int`** . En cambio, los números reales sí tienen parte decimal y el nombre en Python es **`float`** . Elegiremos el tipo que mejor se adapte a cada problema; por ejemplo, si queremos *contar* el número de apariciones de una letra en un texto utilizaremos un entero, pero si se trata de calcular el *porcentaje* de frases que la contienen puede que nos interese un número decimal.

Para ver el tipo de una expresión en todo momento podemos utilizar la función predefinida **`type`** . Así, si tecleamos en una celda de código `type(3)` y ejecutamos dicha casilla el sistema nos mostrará **`int`**, mientras que, si en otra casilla escribimos `type(3.0)`, el resultado de la ejecución será **`float`**. Vemos así que, aunque para nosotros `3` y `3.0` son los mismos números, Python los distingue por su tipo. Sin embargo, esto no significa que no se puedan «mezclar»; por ejemplo, podemos probar `3+4.0` que nos mostrará `7.0`. Obsérvese que el resultado de esta suma es un número real, que es el tipo más general, es decir, internamente Python ha convertido `3` en `3.0` antes de hacer la suma.

Los números permiten hacer todas las operaciones usuales en Python incluyendo operaciones aritméticas como sumar (+), restar (-), multiplicar (*), dividir (/), y obtener el resto de la división (%), tal y como muestra el siguiente ejemplo:

Ejemplo 1.3.1.

```
print( 3+4, 5.1*2.3, 4/3, 5 % 3 )
```

mostrará (omitimos algunos decimales) «`7 11.7299 1.3333 2`». El ejemplo 1.3.1 muestra un par de detalles interesantes más allá del obvio resultado de las operaciones aritméticas:

- Como vemos, los valores mostrados son los resultados de evaluar las operaciones, es decir, la función **`print`** evalúa la operación antes de mostrarla. Esto es un principio general en Python, siempre que se encuentre con una expresión lo primero que hará es reducirla hasta convertirla en un valor simple y esto siempre de «dentro hacia fuera». Por ejemplo, para evaluar `print(3+4)`, Python primero evalúa la expresión interior, lo que da `print(7)`, que finalmente muestra por pantalla el número 7.
- Además, vemos que aquí no estamos utilizando las comillas como sí hacíamos en `print("¡Hola Colab!")` porque los números no se escriben entre comillas. Esto es importante porque si escribimos `print("3+4")` el sistema escribirá `3+4`. Esto parece contradecir el punto anterior porque la expresión no se evalúa, pero

no hay contradicción porque al incluir las comillas hemos dicho al lenguaje que `3+4` es ahora un *string*, esto es, un mensaje, que no necesita ser evaluado.

Tipo lógico

A menudo en Python necesitaremos comprobar si se cumple una condición, por ejemplo si una oración tiene un verbo en presente o no. Para indicar que un enunciado es *cierto* o *falso* se emplea un tipo con solo dos valores: `True` para representar cierto y `False` para representar falso. La mayúscula al principio de ambas constantes es necesaria ya que Python es un lenguaje que distingue las palabras escritas con y sin mayúscula, de forma que `true` será erróneo.

Este tipo se conoce como el *tipo lógico* o *booleano* en honor al matemático y lógico inglés George Boole, que en el siglo XIX estableció los principios de la aritmética computacional a partir de estos valores.

Los valores lógicos nos aparecerán de forma natural cuando empleemos los operadores de comparación o *relacionales*: $<$, $<=$, $>$, $>=$, $==$, $!=$, que representan, respectivamente, las operaciones menor, menor o igual, mayor, mayor o igual, igual y distinto. Por ejemplo:

Ejemplo 1.3.2.

```
print( 3<4, 7!=7 )
```

mostrará «`True False`».

Los valores lógicos se pueden combinar entre sí para dar lugar a expresiones más complejas mediante los *operadores* **and**, **or** y **not** . La primera, **and**, devolverá `True` solo si se combinan dos expresiones y ambas se evalúan `True`, o `False` en caso contrario. Por su parte, el operador **or** devuelve `True` si al menos una de las expresiones que se combinan se evalúa a `True`, y `False` en caso contrario (es decir, si ambas expresiones se reducen a `False`). Finalmente, la negación **not** se aplica a una sola expresión lógica y devuelve lo contrario, esto es, `True` si la expresión se evalúa a `False`, y viceversa. Quizás un pequeño ejemplo ayude a entender mejor el papel de estos operadores:

Ejemplo 1.3.3.

```
print( 3<4 and 4>4, 3<4 or 4>4, not (3>4) )
```

mostrará «`False True True`», porque

– En la primera expresión «`3<4 and 4>4`» no son ciertas ambas expresiones, ya que `3<4` se evalúa a `True` pero 4 no es mayor que 4, es decir, `4>4` se evalúa a `False`.

©RC Libros

- La expresión «3<4 **or** 4>4» se convierte en True, porque al menos una de las dos expresiones (en este caso 3<4) es cierta.
- Por último, «**not** (3>4)» se reduce a True porque 3>4 se reduce a False y **not** lo convierte en True.

Nos quedaría aún un tercer tipo simple por estudiar, los *strings* pero vamos a dejarlos para dentro de un momento cuando hablemos de secuencias.

VARIABLES

Hasta ahora hemos mostrado los valores directamente como parte de las expresiones, pero muy a menudo nos interesará representar un valor mediante un nombre, lo que llamaremos una *variable*. Por ejemplo, tendremos una variable que contiene un texto, otra variable que contiene el texto después de haber extraído sus raíces lexicográficas, etc. Un ejemplo sencillo de variable:

Ejemplo 1.4.1.

```
días=7
print(días)
```

El nombre de la variable lo decide el programador y puede ser cualquier combinación de letras, dígitos y el carácter '_', aunque no pueden comenzar por dígito ni contener espacios. En el ejemplo 1.4.1 días es un nombre válido de variable que representa el valor 7.

Como vemos en el ejemplo, las variables toman su valor mediante el operador de asignación = (que no debemos confundir con el operador de comparación =;=).

Cada vez que Python encuentre un nombre de variable lo evaluará al valor que representa. En el ejemplo anterior, **print**(días) se evalúa a **print**(7) y por tanto muestra un valor 7. Es importante notar que antes de ser evaluada una expresión que contiene variables todas ellas tengan ya asignado un valor (en la misma celda o en celdas anteriores que ya han sido ejecutadas). Por ello, si probamos una instrucción como **print**(tururú) el resultado será un error porque al encontrar tururú sin estar entre comillas, Python asume que es una variable e intenta cambiar el nombre de variable por su valor, pero como no tiene todavía ninguno (diremos que no ha sido *inicializada*) se mostrará un error.

Las variables reciben su nombre de la propiedad de que pueden tomar diferentes valores durante la ejecución de un programa, propiedad muy útil que sin embargo puede resultar al principio un poco confusa. Por ejemplo:

Ejemplo 1.4.2.

```
días=7
print(días)
horas = días *  24
print(horas)
```

mostrará primero 7 y en la siguiente línea 168. ¿Por qué? Las dos primeras líneas ya las hemos analizado; son las del ejemplo 1.4.1, fijémonos ahora en la tercera línea, la más compleja, `horas = días * 24`. Al ver la primera parte `horas =` Python entiende que queremos guardar un valor en la variable `horas` (que ya tenga uno o no es indiferente), por lo que necesita evaluar el resto, es decir, `días * 24`, que sustituyendo el valor de *días* es `7 * 24` que se reduce finalmente a `168`, valor que se hace corresponder con `horas`.

Si se ha entendido correctamente este ejemplo, el que aparece a continuación también debe resultar inteligible a pesar de su apariencia más compleja:

Ejemplo 1.4.3.

```
días=7
print(días)
días = días + 1
print(días)
```

Aquí la diferencia está en que en la tercera línea `días = días + 1` se utiliza la misma variable a ambos lados, pero la lógica es la misma: como `días =` es una asignación, Python evaluará la parte derecha `días + 1`, que equivale a 7+1, y guardará en `días` el resultado, el valor 8. El efecto final de la instrucción es que hemos incrementado el valor de la variable `días` en 1.

SECUENCIAS

Los enteros, reales, booleanos o caracteres son datos simples. Sin embargo, los lenguajes de programación ofrecen también la posibilidad de agregar varios de estos datos simples formando estructuras que pueden manejarse como una unidad: almacenarse en variables, mostrarse por pantalla, etc. En Python tenemos tres estructuras de tipo secuencia fundamentales:

- *Strings*: secuencias de caracteres entre comillas.
- *Listas*: elementos entre corchetes y separados por comas, una estructura muy potente y flexible que vamos a utilizar a lo largo de todo el texto.
- *Tuplas*: secuencias de elementos entre paréntesis separadas entre comas, similares en apariencia a las listas, pero con la diferencia de que no se pueden modificar.

©RC Libros