CAPÍTULO 3

Tipos de datos

Igual que en el mundo real cada objeto pertenece a una categoría, en programación manejamos objetos que tienen asociado un tipo determinado. En este capítulo se verán los tipos de datos básicos con los que podemos trabajar en Python.

3.1 Datos

Los programas están formados por **código** y **datos**. Pero a nivel interno de la memoria del ordenador no son más que una secuencia de bits. La interpretación de estos bits depende del lenguaje de programación, que almacena en la memoria no sólo el puro dato sino distintos metadatos.[1]

Cada «trozo» de memoria contiene realmente un objeto, de ahí que se diga que en Python **todo son objetos**. Y cada objeto tiene, al menos, los siguientes campos:

- Un **tipo** del dato almacenado.
- Un **identificador** único para distinguirlo de otros objetos.
- Un **valor** consistente con su tipo.

3.1.1 Tipos de datos

A continuación se muestran los distintos tipos de datos que podemos encontrar en Python, sin incluir aquellos que proveen paquetes externos:

[1] Foto original de portada por Alexander Sinn en Unsplash.

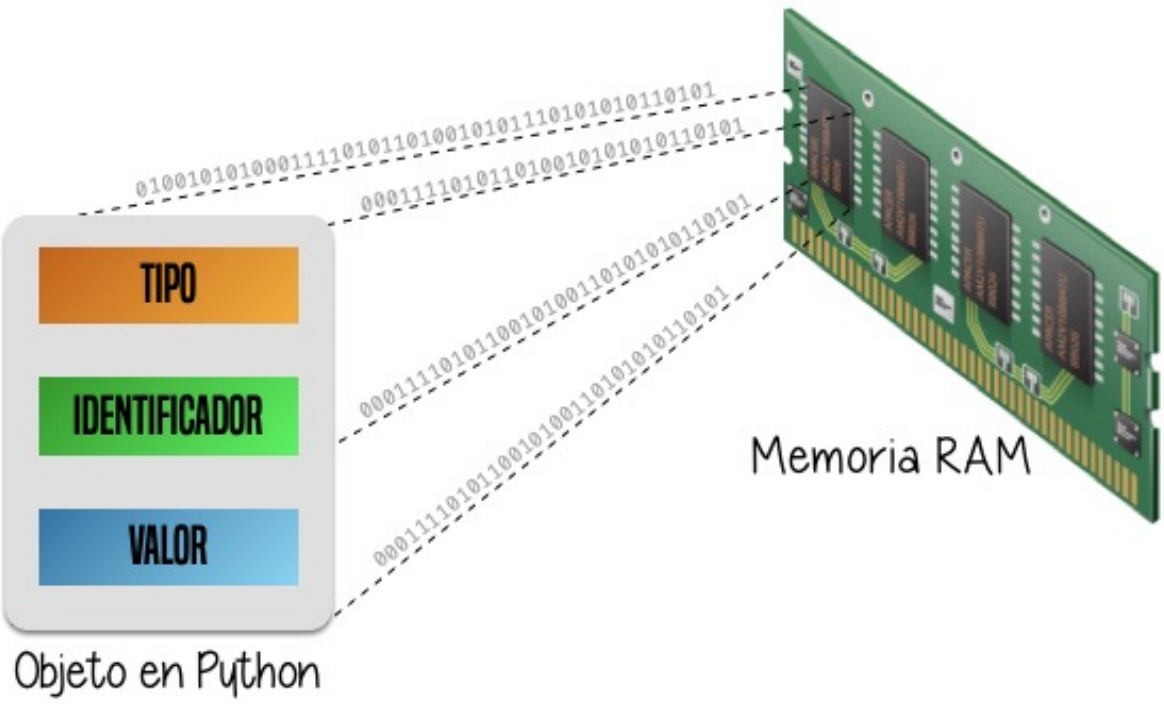

Figura 1: Esquema (*metadatos*) de un objeto en Python

Tabla 1: Tipos de datos en Python

Nombre	Tipo	Ejemplos
Booleano	bool	`True`, `False`
Entero	int	`21`, `34500`, `34_500`
Flotante	float	`3.14`, `1.5e3`
Complejo	complex	`2j`, `3 + 5j`
Cadena	str	`'tfn'`, `'''tenerife - islas canarias'''`
Tupla	tuple	`(1, 3, 5)`
Lista	list	`['Chrome', 'Firefox']`
Conjunto	set	`set([2, 4, 6])`
Diccionario	dict	`{'Chrome': 'v79' , 'Firefox': 'v71'}`

3.1.2 Variables

Las **variables** son fundamentales ya que permiten definir **nombres** para los **valores** que tenemos en memoria y que vamos a usar en nuestro programa.

Figura 2: Uso de un *nombre* de variable

Reglas para nombrar variables

En Python existen una serie de reglas para los nombres de variables:

1. **Sólo pueden contener los siguientes caracteres**[2]**:**
    - Letras minúsculas.
    - Letras mayúsculas.
    - Dígitos.
    - Guiones bajos (_).
2. Deben **empezar con una letra o un guión bajo**, nunca con un dígito.
3. No pueden ser una **palabra reservada** del lenguaje («keywords»).

[2] Para ser exactos, sí se pueden utilizar otros caracteres, e incluso *emojis* en los nombres de variables, aunque no suele ser una práctica extendida, ya que podría dificultar la legibilidad.

Podemos obtener un listado de las palabras reservadas del lenguaje de la siguiente forma:

```
>>> help('keywords')

Here is a list of the Python keywords.  Enter any keyword to get more help.

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

Nota: Por lo general se prefiere dar nombres **en inglés** a las variables que utilicemos, ya que así hacemos nuestro código más «internacional» y con la posibilidad de que otras personas puedan leerlo, entenderlo y – llegado el caso – modificarlo. Es sólo una recomendación, nada impide que se haga en castellano.

Importante: Los nombres de variables son «case-sensitive»[3]. Por ejemplo, `stuff` y `Stuff` son nombres diferentes.

Ejemplos de nombres de variables

Veamos a continuación una tabla con nombres de variables:

Tabla 2: Ejemplos de nombres de variables

Válido	Inválido	Razón
`a`	`3`	Empieza por un dígito
`a3`	`3a`	Empieza por un dígito
`a_b_c___95`	`another-name`	Contiene un caracter no permitido
`_abc`	`with`	Es una palabra reservada del lenguaje
`_3a`	`3_a`	Empieza por un dígito

[3] Sensible a cambios en mayúsculas y minúsculas.

Convenciones para nombres

Mientras se sigan las *reglas* que hemos visto para nombrar variables no hay problema en la forma en la que se escriban, pero sí existe una convención para la **nomenclatura de las variables**. Se utiliza el llamado `snake_case` en el que utilizamos **caracteres en minúsculas** (incluyendo dígitos si procede) junto con **guiones bajos** – cuando sean necesarios para su legibilidad –.[4] Por ejemplo, para nombrar una variable que almacene el número de canciones en nuestro ordenador, podríamos usar `num_songs`.

Esta convención, y muchas otras, están definidas en un documento denominado PEP 8. Se trata de una **guía de estilo** para escribir código en Python. Los PEPs[5] son las propuestas que se hacen para la mejora del lenguaje.

Aunque hay múltiples herramientas disponibles para la comprobación del estilo de código, una bastante accesible es http://pep8online.com/ ya que no necesita instalación, simplemente pegar nuestro código y verificar.

Constantes

Un caso especial y que vale la pena destacar son las **constantes**. Podríamos decir que es un tipo de variable pero que su valor no cambia a lo largo de nuestro programa. Por ejemplo la velocidad de la luz. Sabemos que su valor es constante de 300.000 km/s. En el caso de las constantes utilizamos **mayúsculas** (incluyendo guiones bajos si es necesario) para nombrarlas. Para la velocidad de la luz nuestra constante se podría llamar: `LIGHT_SPEED`.

Elegir buenos nombres

Se suele decir que una persona programadora (con cierta experiencia), a lo que dedica más tiempo, es a buscar un buen nombre para sus variables. Quizás pueda resultar algo excesivo pero da una idea de lo importante que es esta tarea. Es fundamental que los nombres de variables sean **autoexplicativos**, pero siempre llegando a un compromiso entre ser concisos y claros.

Supongamos que queremos buscar un nombre de variable para almacenar el número de elementos que se deben manejar en un pedido:

1. `n`
2. `num_elements`
3. `number_of_elements`
4. `number_of_elements_to_be_handled`

[4] Más información sobre convenciones de nombres en PEP 8.
[5] Del término inglés «Python Enhancement Proposals».

No existe una regla mágica que nos diga cuál es el nombre perfecto, pero podemos aplicar el *sentido común* y, a través de la experiencia, ir detectando aquellos nombres que sean más adecuados. En el ejemplo anterior, quizás podríamos descartar de principio la opción *1* y la *4* (por ser demasiado cortas o demasiado largas); nos quedaríamos con las otras dos. Si nos fijamos bien, casi no hay mucha información adicional de la opción *3* con respecto a la *2*. Así que podríamos concluir que la opción *2* es válida para nuestras necesidades. En cualquier caso esto dependerá siempre del contexto del problema que estemos tratando.

Como regla general:

- Usar **nombres** para *variables* (ejemplo `article`).
- Usar **verbos** para *funciones* (ejemplo `get_article()`).
- Usar **adjetivos** para *booleanos* (ejemplo `available`).

3.1.3 Asignación

En Python se usa el símbolo `=` para **asignar** un valor a una variable:

Figura 3: Asignación de *valor* a *nombre* de variable

Nota: Hay que diferenciar la asignación en Python con la igualación en matemáticas. El símbolo `=` lo hemos aprendido desde siempre como una *equivalencia* entre dos *expresiones algebraicas*, sin embargo en Python nos indica una *sentencia de asignación*, del valor (en la derecha) al nombre (en la izquierda).

Algunos ejemplos de asignaciones a *variables*:

```
>>> total_population = 157503
>>> avg_temperature = 16.8
>>> city_name = 'San Cristóbal de La Laguna'
```

Algunos ejemplos de asignaciones a *constantes*:

```
>>> SOUND_SPEED = 343.2
>>> WATER_DENSITY = 997
>>> EARTH_NAME = 'La Tierra'
```

Python nos ofrece la posibilidad de hacer una **asignación múltiple** de la siguiente manera:

```
>>> tres = three = drei = 3
```

En este caso las tres variables utilizadas en el «lado izquierdo» tomarán el valor `3`.

Recordemos que los nombres de variables deben seguir unas *reglas establecidas*, de lo contrario obtendremos un **error sintáctico** del intérprete de Python:

```
>>> 7floor = 40  # el nombre empieza por un dígito
  File "<stdin>", line 1
    7floor = 40
         ^
SyntaxError: invalid syntax

>>> for = 'Bucle'  # el nombre usa la palabra reservada "for"
  File "<stdin>", line 1
    for = 'Bucle'
        ^
SyntaxError: invalid syntax

>>> screen-size = 14  # el nombre usa un caracter no válido
  File "<stdin>", line 1
SyntaxError: can't assign to operator
```

Asignando una variable a otra variable

Las asignaciones que hemos hecho hasta ahora han sido de un **valor literal** a una variable. Pero nada impide que podamos hacer asignaciones de una variable a otra variable:

```
>>> people = 157503
>>> total_population = people
>>> total_population
157503
```

Eso sí, la variable que utilicemos como valor de asignación **debe existir previamente**, ya que si no es así, obtendremos un error informando de que no está definida:

```
>>> total_population = lot_of_people
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'lot_of_people' is not defined
```

De hecho, en el *lado derecho* de la asignación pueden aparecer *expresiones* más complejas que se verán en su momento.

Conocer el valor de una variable

Hemos visto previamente cómo asignar un valor a una variable, pero aún no sabemos cómo «comprobar» el valor que tiene dicha variable. Para ello podemos utilizar dos estrategias:

1. Si estamos en una «shell» de Python, basta con que usemos el nombre de la variable:

```
>>> final_stock = 38934
>>> final_stock
38934
```

2. Si estamos escribiendo un programa desde el editor, podemos hacer uso de `print`:

```
final_stock = 38934
print(final_stock)
```

Nota: `print` sirve también cuando estamos en una sesión interactiva de Python («shell»)

Conocer el tipo de una variable

Para poder descubrir el tipo de un literal o una variable, Python nos ofrece la función `type()`. Veamos algunos ejemplos de su uso:

```
>>> type(9)
int

>>> type(1.2)
float

>>> height = 3718
>>> type(height)
int

>>> sound_speed = 343.2
>>> type(sound_speed)
float
```

Ejercicio

1. Asigna un valor entero `2001` a la variable `space_odyssey` y muestra su valor.

2. Descubre el tipo del literal `'Good night & Good luck'`.
3. Identifica el tipo del literal `True`.
4. Asigna la expresión `10 * 3.0` a la variable `result` y muestra su tipo.

3.1.4 Mutabilidad

Nivel avanzado

Las variables son nombres, no lugares. Detrás de esta frase se esconde la reflexión de que cuando asignamos un valor a una variable, lo que realmente está ocurriendo es que se hace **apuntar** el nombre de la variable a una zona de memoria en el que se representa el objeto (con su valor).

Si realizamos la asignación de una variable a un valor lo que está ocurriendo es que el nombre de la variable es una **referencia** al valor, no el valor en sí mismo:

```
>>> a = 5
```

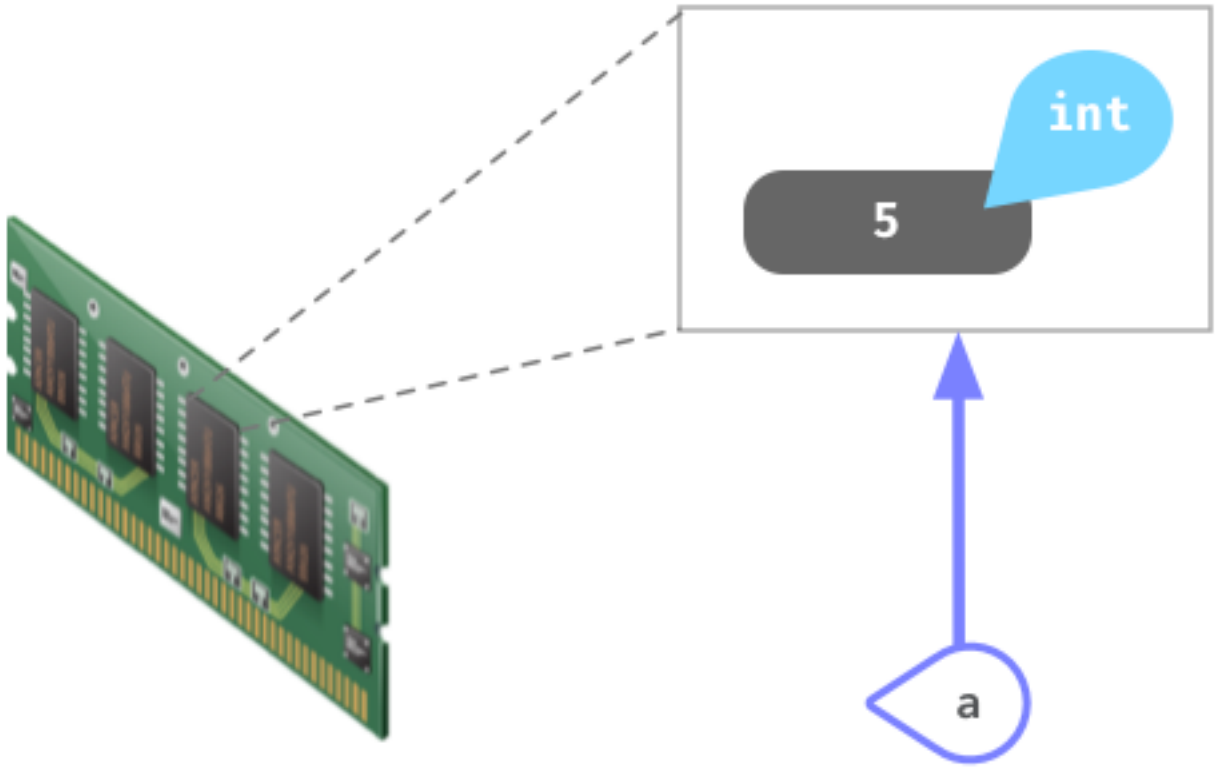

Figura 4: Representación de la asignación de valor a variable

Si ahora «copiamos» el valor de `a` en otra variable `b` se podría esperar que hubiera otro espacio en memoria para dicho valor, pero como ya hemos dicho, son referencias a memoria:

```
>>> b = a
```

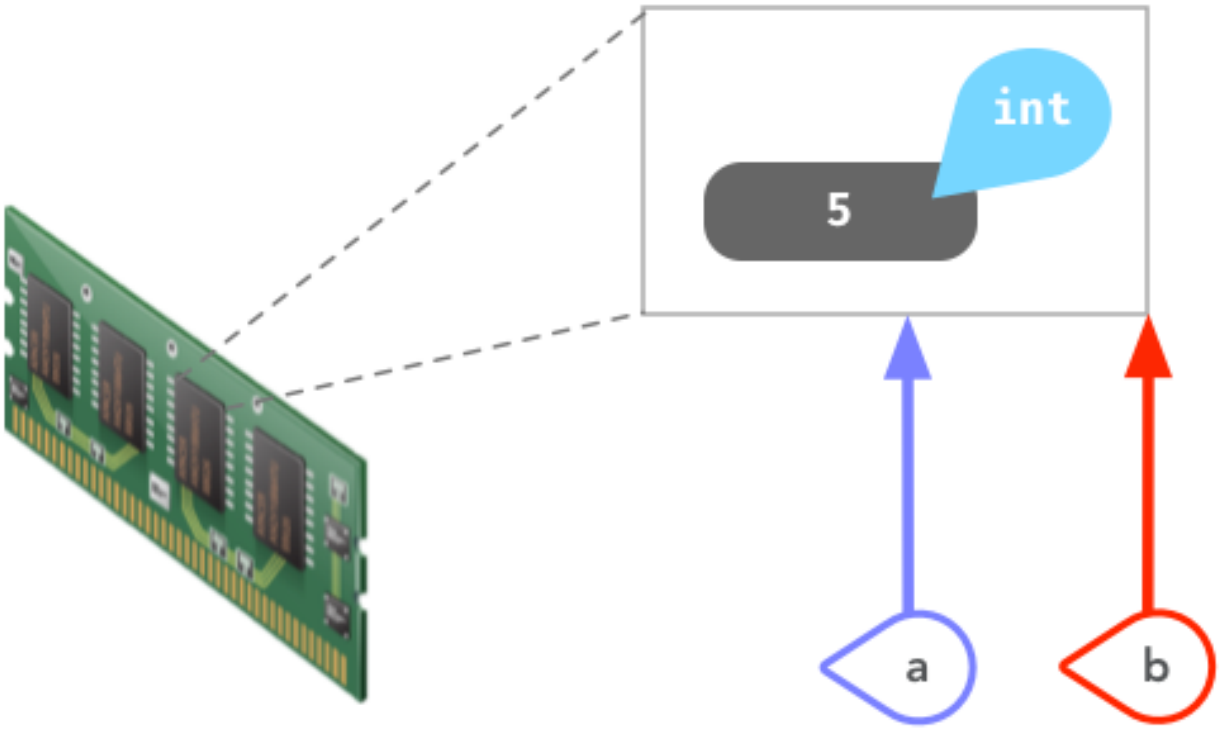

Figura 5: Representación de la asignación de una variable a otra variable

La función `id()` nos permite conocer la dirección de memoria[6] de un objeto en Python. A través de ella podemos comprobar que los dos objetos que hemos creado «apuntan» a la misma zona de memoria:

```
>>> id(a)
4445989712

>>> id(b)
4445989712
```

¿Y esto qué tiene que ver con la **mutabilidad**? Pues se dice, por ejemplo, que un **entero** es **inmutable** ya que a la hora de modificar su valor obtenemos una nueva *zona de memoria*, o lo que es lo mismo, un nuevo objeto:

```
>>> a = 5
>>> id(a)
```

(continué en la próxima página)

[6] Esto es un detalle de implementación de CPython.

(proviene de la página anterior)

```
4310690224

>>> a = 7
>>> id(a)
4310690288
```

Sin embargo, si tratamos con **listas**, podemos ver que la modificación de alguno de sus valores no implica un cambio en la posición de memoria de la variable, por lo que se habla de objetos **mutables**.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/lvCyXeL

La característica de que los nombres de variables sean referencias a objetos en memoria es la que hace posible diferenciar entre **objetos mutables e inmutables**:

Inmutable	Mutable
`bool`	`list`
`int`	`set`
`float`	`dict`
`str`	
`tuple`	

Importante: El hecho de que un tipo de datos sea inmutable significa que no podemos modificar su valor «in-situ», pero siempre podremos asignarle un nuevo valor (hacerlo apuntar a otra zona de memoria).

3.1.5 Funciones «built-in»

Nivel intermedio

Hemos ido usando una serie de *funciones* sin ser especialmente conscientes de ello. Esto se debe a que son funciones «built-in» o incorporadas por defecto en el propio lenguaje Python.

Tabla 3: Funciones «built-in»

`abs()`	`delattr()`	`hash()`	`memoryview()`	`set()`
`all()`	`dict()`	`help()`	`min()`	`setattr()`
`any()`	`any()`	`hex()`	`next()`	`slice()`
`ascii()`	`divmod()`	`id()`	`object()`	`sorted()`
`bin()`	`enumerate()`	`input()`	`oct()`	`staticmethod()`

continué en la próxima página

Tabla 3 – proviene de la página anterior

bool()	eval()	int()	open()	str()
breakpoint()	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()	

Los detalles de estas funciones se puede consultar en la documentación oficial de Python.

AMPLIAR CONOCIMIENTOS

- Basic Data Types in Python
- Variables in Python
- Immutability in Python

3.2 Números

En esta sección veremos los tipos de datos númericos que ofrece Python centrándonos en **booleanos**, **enteros** y **flotantes**.[1]

3.2.1 Booleanos

George Boole es considerado como uno de los fundadores del campo de las ciencias de la computación y fue el creador del Álgebra de Boole que da lugar, entre otras estructuras algebraicas, a la Lógica binaria. En esta lógica las variables sólo pueden tomar dos valores discretos: **verdadero** o **falso**.

El tipo de datos `bool` proviene de lo explicado anteriormente y admite dos posibles valores:

- `True` que se corresponde con *verdadero* (y también con **1** en su representación numérica).
- `False` que se corresponde con *falso* (y también con **0** en su representación numérica).

Veamos un ejemplo de su uso:

```
>>> is_opened = True
>>> is_opened
True

>>> has_sugar = False
>>> has_sugar
False
```

La primera variable `is_opened` está representando el hecho de que algo esté abierto, y al tomar el valor `True` podemos concluir que sí. La segunda variable `has_sugar` nos indica si una bebida tiene azúcar; dado que toma el valor `False` inferimos que no lleva azúcar.

> **Atención:** Tal y como se explicó en *este apartado*, los nombres de variables son «case-sensitive». De igual modo el tipo booleano toma valores `True` y `False` con **la primera letra en mayúsculas**. De no ser así obtendríamos un error sintáctico.

```
>>> is_opened = true
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
>>> has_sugar = false
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'false' is not defined
```

[1] Foto original de portada por Brett Jordan en Unsplash.

3.2.2 Enteros

Los números enteros no tienen decimales pero sí pueden contener signo y estar expresados en alguna base distinta de la usual (base 10).

Literales enteros

Veamos algunos ejemplos de números enteros:

```
>>> 8
8
>>> 0
0
>>> 08
  File "<stdin>", line 1
    08
     ^
SyntaxError: invalid token
>>> 99
99
>>> +99
99
>>> -99
-99
>>> 3000000
3000000
>>> 3_000_000
3000000
```

Dos detalles a tener en cuenta:

- No podemos comenzar un número entero por `0`.
- Python permite dividir los números enteros con *guiones bajos* `_` para clarificar su lectura/escritura. A efectos prácticos es como si esos guiones bajos no existieran.

Operaciones con enteros

A continuación se muestra una tabla con las distintas operaciones sobre enteros que podemos realizar en Python:

Tabla 4: Operaciones con enteros en Python

Operador	Descripción	Ejemplo	Resultado
+	Suma	`3 + 9`	`12`

continué en la próxima página

Tabla 4 – proviene de la página anterior

Operador	Descripción	Ejemplo	Resultado
-	Resta	6 - 2	4
*	Multiplicación	5 * 5	25
/	División flotante	9 / 2	4.5
//	División entera	9 // 2	4
%	Módulo	9 % 4	1
**	Exponenciación	2 ** 4	16

Veamos algunas pruebas de estos operadores:

```
>>> 2 + 8 + 4
14
>>> 4 ** 4
256
>>> 7 / 3
2.3333333333333335
>>> 7 // 3
2
>>> 6 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Es de buen estilo de programación **dejar un espacio** entre cada operador. Además hay que tener en cuenta que podemos obtener errores dependiendo de la operación (más bien de los *operandos*) que estemos utilizando, como es el caso de la *división por cero*.

Asignación aumentada

Python nos ofrece la posibilidad de escribir una asignación aumentada mezclando la *asignación* y un *operador*.

Figura 6: Asignación aumentada en Python

Supongamos que disponemos de 100 vehículos en stock y que durante el pasado mes se han

vendido 20 de ellos. Veamos cómo sería el código con asignación tradicional vs. asignación aumentada:

Lista 1: Asignación tradicional

```
>>> total_cars = 100
>>> sold_cars = 20
>>> total_cars = total_cars - sold_cars
>>> total_cars
80
```

Lista 2: Asignación aumentada

```
>>> total_cars = 100
>>> sold_cars = 20
>>> total_cars -= sold_cars
>>> total_cars
80
```

Estas dos formas son equivalentes a nivel de resultados y funcionalidad, pero obviamente tienen diferencias de escritura y legibilidad. De este mismo modo, podemos aplicar un formato compacto al resto de operaciones:

```
>>> random_number = 15

>>> random_number += 5
>>> random_number
20

>>> random_number *= 3
>>> random_number
60

>>> random_number //= 4
>>> random_number
15

>>> random_number **= 1
>>> random_number
15
```

Módulo

La operación **módulo** (también llamado **resto**), cuyo símbolo en Python es `%`, se define como el resto de dividir dos números. Veamos un ejemplo para enteder bien su funcionamiento:

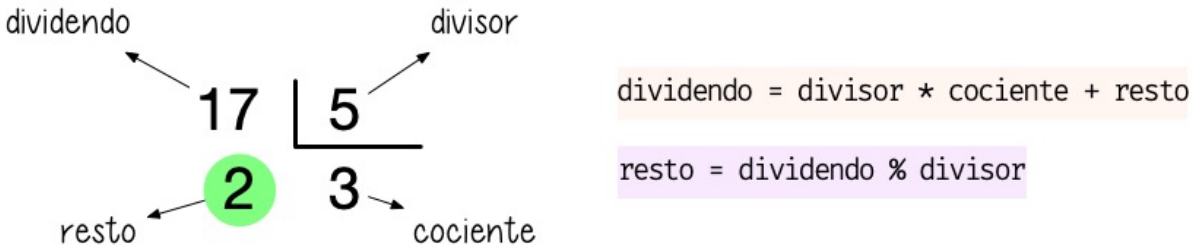

Figura 7: Operador «módulo» en Python

```
>>> dividendo = 17
>>> divisor = 5

>>> cociente = dividendo // divisor  # división entera
>>> resto = dividendo % divisor

>>> cociente
3
>>> resto
2
```

Exponenciación

Para elevar un número a otro en Python utilizamos el operador de exponenciación `**`:

```
>>> 4 ** 3
64
>>> 4 * 4 * 4
64
```

Se debe tener en cuenta que también podemos elevar un número entero a un **número decimal**. En este caso es como si estuviéramos haciendo una *raíz*[2]. Por ejemplo:

$$4^{\frac{1}{2}} = 4^{0.5} = \sqrt{4} = 2$$

Hecho en Python:

[2] No siempre es una raíz cuadrada porque se invierten numerador y denominador.

```
>>> 4 ** 0.5
2.0
```

Ejercicio

Una ecuación de segundo grado se define como $ax^2 + bx + c = 0$, y (en determinados casos) tiene dos soluciones:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$
$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Dados los coeficientes `a=4`, `b=-6` y `c=2` calcule sus dos soluciones. Tenga en cuenta subdividir los cálculos y reutilizar variables (por ejemplo el discriminante).

La solución para los valores anteriores es:

- `x1 = 1`
- `x2 = 0.5`

Recuerde que la **raíz cuadrada** se puede calcular como la exponenciación a $\frac{1}{2}$.

Puede comprobar los resultados para otros valores de entrada con esta aplicación para resolver ecuaciones cuadráticas.

Valor absoluto

Python ofrece la función `abs()` para obtener el valor absoluto de un número:

```
>>> abs(-1)
1

>>> abs(1)
1

>>> abs(-3.14)
3.14

>>> abs(3.14)
3.14
```

Límite de un entero

Nivel avanzado

¿Cómo de grande puede ser un `int` en Python? La respuesta es **de cualquier tamaño**. Por poner un ejemplo, supongamos que queremos representar un centillón. Este valor viene a ser un «1» seguido por ¡600 ceros! ¿Será capaz Python de almacenarlo?

```
>>> centillion = 10 ** 600

>>> centillion
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Nota: En muchos lenguajes tratar con enteros tan largos causaría un «integer overflow». No es el caso de Python que puede manejar estos valores sin problema.

3.2.3 Flotantes

Los números en **punto flotante**[3] tienen **parte decimal**. Veamos algunos ejemplos de flotantes en Python.

[3] Punto o coma flotante es una notación científica usada por computadores.

Lista 3: Distintas formas de escribir el flotante *4.0*

```
>>> 4.0
4.0
>>> 4.
4.0
>>> 04.0
4.0
>>> 04.
4.0
>>> 4.000_000
4.0
>>> 4e0  # 4.0 * (10 ** 0)
4.0
```

Conversión de tipos

El hecho de que existan distintos tipos de datos en Python (y en el resto de lenguajes de programación) es una ventaja a la hora de representar la información del mundo real de la mejor manera posible. Pero también se hace necesario buscar mecanismos para convertir unos tipos de datos en otros.

Conversión implícita

Cuando mezclamos enteros, booleanos y flotantes, Python realiza automáticamente una conversión implícita (o **promoción**) de los valores al tipo de «mayor rango». Veamos algunos ejemplos de esto:

```
>>> True + 25
26
>>> 7 * False
0
>>> True + False
1
>>> 21.8 + True
22.8
>>> 10 + 11.3
21.3
```

Podemos resumir la conversión implícita en la siguiente tabla:

Tipo 1	Tipo 2	Resultado
`bool`	`int`	`int`
`bool`	`float`	`float`
`int`	`float`	`float`

Se puede ver claramente que la conversión numérica de los valores booleanos es:

- `True` 👉 `1`
- `False` 👉 `0`

Conversión explícita

Aunque más adelante veremos el concepto de **función**, desde ahora podemos decir que existen una serie de funciones para realizar conversiones explícitas de un tipo a otro:

`bool()` Convierte el tipo a *booleano*.

`int()` Convierte el tipo a *entero*.

`float()` Convierte el tipo a *flotante*.

Veamos algunos ejemplos de estas funciones:

```
>>> bool(1)
True
>>> bool(0)
False
>>> int(True)
1
>>> int(False)
0
>>> float(1)
1.0
>>> float(0)
0.0
>>> float(True)
1.0
>>> float(False)
0.0
```

En el caso de que usemos la función `int()` sobre un valor flotante, nos retorna su **parte baja**:

$$int(x) = \lfloor x \rfloor$$

Por ejemplo:

```
>>> int(3.1)
3
>>> int(3.5)
3
>>> int(3.9)
3
```

Para **obtener el tipo** de una variable podemos hacer uso de la función `type()`:

```
>>> is_raining = False

>>> type(is_raining)
<class 'bool'>

>>> sound_level = 35
>>> type(sound_level)
<class 'int'>

>>> temperature = 36.6
>>> type(temperature)
<class 'float'>
```

Igualmente existe la posibilidad de **comprobar el tipo** que tiene una variable mediante la función `isinstance()`:

```
>>> isinstance(is_raining, bool)
True
>>> isinstance(sound_level, int)
True
>>> isinstance(temperature, float)
True
```

Ejercicio

Existe una aproximación al seno de un ángulo x expresado en *grados*:

$$sin(x) \approx \frac{4x(180 - x)}{40500 - x(180 - x)}$$

Calcule dicha aproximación utilizando operaciones en Python. Descomponga la expresión en subcálculos almacenados en variables. Tenga en cuenta aquellas expresiones comunes para no repetir cálculos y seguir el principio DRY.

¿Qué tal funciona la aproximación? Compare sus resultados con estos:

- $sin(90) = 1.0$
- $sin(45) = 0.7071067811865475$

- $sin(50) = 0.766044443118978$

Errores de aproximación

Nivel intermedio

Supongamos el siguiente cálculo:

```
>>> (19 / 155) * (155 / 19)
    0.9999999999999999
```

Debería dar 1.0, pero no es así puesto que la representación interna de los valores en **coma flotante** sigue el estándar IEEE 754 y estamos trabajando con aritmética finita.

Aunque existen distintas formas de solventar esta limitación, de momento veremos una de las más sencillas utilizando la función «built-in» round() que nos permite redondear un número flotante a un número determinado de decimales:

```
>>> pi = 3.14159265359

>>> round(pi)
3
>>> round(pi, 1)
3.1
>>> round(pi, 2)
3.14
>>> round(pi, 3)
3.142
>>> round(pi, 4)
3.1416
>>> round(pi, 5)
3.14159
```

Para el caso del error de aproximación que nos ocupa:

```
>>> result = (19 / 155) * (155 / 19)

>>> round(result, 1)
1.0
```

Prudencia: `round()` aproxima al valor más cercano, mientras que `int()` obtiene siepre el entero «por abajo».

Límite de un flotante

A diferencia de los *enteros*, los números flotantes sí que tienen un límite en Python. Para descubrirlo podemos ejecutar el siguiente código:

```
>>> import sys

>>> sys.float_info.min
2.2250738585072014e-308

>>> sys.float_info.max
1.7976931348623157e+308
```

3.2.4 Bases

Nivel intermedio

Los valores numéricos con los que estamos acostumbrados a trabajar están en **base 10** (o decimal). Esto indica que disponemos de 10 «símbolos» para representar las cantidades. En este caso del `0` al `9`.

Pero también es posible representar números en **otras bases**. Python nos ofrece una serie de **prefijos** y **funciones** para este cometido.

Base binaria

Cuenta con **2** símbolos para representar los valores: `0` y `1`.

Prefijo: `0b`

```
>>> 0b1001
9
>>> 0b1100
12
```

Función: `bin()`

```
>>> bin(9)
'0b1001'
>>> bin(12)
'0b1100'
```

Base octal

Cuenta con **8** símbolos para representar los valores: `0`, `1`, `2`, `3`, `4`, `5`, `6` y `7`.

Prefijo: `0o`

```
>>> 0o6243
3235
>>> 0o1257
687
```

Función: `oct()`

```
>>> oct(3235)
'0o6243'
>>> oct(687)
'0o1257'
```

Base hexadecimal

Cuenta con **16** símbolos para representar los valores: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E` y `F`.

Prefijo: `0x`

```
>>> 0x7F2A
32554
>>> 0x48FF
18687
```

Función: `hex()`

```
>>> hex(32554)
'0x7f2a'
>>> hex(18687)
'0x48ff'
```

Nota: Las letras para la representación hexadecimal no atienden a mayúsculas y minúsculas.

EJERCICIOS DE REPASO

1. Escriba un programa en Python que acepte el radio de un círculo y compute su área (`solución`).

   Entrada: 5
   Salida: 78.5

2. Escriba un programa en Python que acepte el radio de una esfera y obtenga su volumen (`solución`).

   Entrada: 5
   Salida: 523.3333333333334

3. Escriba un programa en Python que acepte la base y la altura de un triángulo y compute su área (`solución`).

   Entrada: base=4; altura=5
   Salida: 10.0

4. Escriba un programa en Python que compute el futuro valor de una cantidad de dinero, a partir del capital inicial, el tipo de interés y el número de años (`solución`).

   Entrada: capital=10000; interés=3.5; años=7
   Salida: 12722.792627665729

5. Escriba un programa en Python que calcule la distancia euclídea entre dos puntos (x_1, y_1) y (x_2, y_2) (`solución`).

   Entrada: $(x_1 = 3, y_1 = 5)$; $(x_2 = -7, y_2 = -4)$
   Salida: 13.45362404707371

AMPLIAR CONOCIMIENTOS

- The Python Square Root Function
- How to Round Numbers in Python
- Operators and Expressions in Python

3.3 Cadenas de texto

Las cadenas de texto son **secuencias** de **caracteres**. También se les conoce como «strings» y nos permiten almacenar información textual de forma muy cómoda.[1]

Es importante destacar que Python 3 almacena los caracteres codificados en el estándar Unicode, lo que es una gran ventaja con respecto a versiones antiguas del lenguaje. Además permite representar una cantidad ingente de símbolos incluyendo los famosos emojis 😎.

3.3.1 Creando «strings»

Para escribir una cadena de texto en Python basta con rodear los caracteres con comillas simples[6]:

```
>>> 'Mi primera cadena en Python'
'Mi primera cadena en Python'
```

Para incluir *comillas dobles* dentro de la cadena de texto no hay mayor inconveniente:

```
>>> 'Los llamados "strings" son secuencias de caracteres'
'Los llamados "strings" son secuencias de caracteres'
```

[1] Foto original de portada por Roman Kraft en Unsplash.

[6] También es posible utilizar comillas dobles. Yo me he decantado por las comillas simples ya que quedan más limpias y suele ser el formato que devuelve el propio intérprete de Python.

Puede surgir la duda de cómo incluimos *comillas simples* dentro de la propia cadena de texto. Veamos soluciones para ello:

Lista 4: Comillas simples escapadas

```
>>> 'Los llamados \'strings\' son secuencias de caracteres'
"Los llamados 'strings' son secuencias de caracteres"
```

Lista 5: Comillas simples dentro de comillas dobles

```
>>> "Los llamados 'strings' son secuencias de caracteres"
"Los llamados 'strings' son secuencias de caracteres"
```

En la primera opción estamos **escapando** las comillas simples para que no sean tratadas como caracteres especiales. En la segunda opción estamos creando el «string» con comillas dobles (por fuera) para poder incluir directamente las comillas simples (por dentro). Python también nos ofrece esta posibilidad.

Comillas triples

Hay una forma alternativa de crear cadenas de texto utilizando *comillas triples*. Su uso está pensado principalmente para **cadenas multilínea**:

```
>>> poem = '''To be, or not to be, that is the question:
... Whether 'tis nobler in the mind to suffer
... The slings and arrows of outrageous fortune,
... Or to take arms against a sea of troubles'''
```

Importante: Los tres puntos ... que aparecen a la izquierda de las líneas no están incluidos en la cadena de texto. Es el símbolo que ofrece el intérprete de Python cuando saltamos de línea.

Cadena vacía

La cadena vacía es aquella que no contiene ningún caracter. Aunque a priori no lo pueda parecer, es un recurso importante en cualquier código. Su representación en Python es la siguiente:

```
>>> ''
''
```

3.3.2 Conversión

Podemos crear «strings» a partir de otros tipos de datos usando la función `str()`:

```
>>> str(True)
'True'
>>> str(10)
'10'
>>> str(21.7)
'21.7'
```

3.3.3 Secuencias de escape

Python permite **escapar** el significado de algunos caracteres para conseguir otros resultados. Si escribimos una barra invertida \ antes del caracter en cuestión, le otorgamos un significado especial.

Quizás la *secuencia de escape* más conocida es `\n` que representa un *salto de línea*, pero existen muchas otras:

```
# Salto de línea
>>> msg = 'Primera línea\nSegunda línea\nTercera línea'
>>> print(msg)
Primera línea
Segunda línea
Tercera línea

# Tabulador
>>> msg = 'Valor = \t40'
>>> print(msg)
Valor =     40

# Comilla simple
>>> msg = 'Necesitamos \'escapar\' la comilla simple'
>>> print(msg)
Necesitamos 'escapar' la comilla simple

# Barra invertida
>>> msg = 'Capítulo \\ Sección \\ Encabezado'
>>> print(msg)
Capítulo \ Sección \ Encabezado
```

Nota: Al utilizar la función `print()` es cuando vemos realmente el resultado de utilizar los caracteres escapados.

Expresiones literales

Nivel intermedio

Hay situaciones en las que nos interesa que los caracteres especiales pierdan ese significado y poder usarlos de otra manera. Existe un modificar de cadena que proporciona Python para tratar el texto *en bruto*. Es el llamado «raw data» y se aplica anteponiendo una `r` a la cadena de texto.

Veamos algunos ejemplos:

```
>>> text = 'abc\ndef'
>>> print(text)
abc
def

>>> text = r'abc\ndef'
>>> print(text)
abc\ndef

>>> text = 'a\tb\tc'
>>> print(text)
a    b    c

>>> text = r'a\tb\tc'
>>> print(text)
a\tb\tc
```

Consejo: El modificador `r''` es muy utilizado para la escritura de **expresiones regulares**.

3.3.4 Más sobre `print()`

Hemos estado utilizando la función `print()` de forma sencilla, pero admite algunos parámetros interesantes:

```
1  >>> msg1 = '¿Sabes por qué estoy acá?'
2  >>> msg2 = 'Porque me apasiona'
3
4  >>> print(msg1, msg2)
5  ¿Sabes por qué estoy acá? Porque me apasiona
6
7  >>> print(msg1, msg2, sep='|')
8  ¿Sabes por qué estoy acá?|Porque me apasiona
9
```

(continué en la próxima página)

(proviene de la página anterior)

```
10 >>> print(msg2, end='!!')
11 Porque me apasiona!!
```

Línea 4: Podemos imprimir todas las variables que queramos separándolas por comas.

Línea 7: El *separador por defecto* entre las variables es un *espacio*, podemos cambiar el caracter que se utiliza como separador entre cadenas.

Línea 10: El *carácter de final de texto* es un *salto de línea*, podemos cambiar el caracter que se utiliza como final de texto.

3.3.5 Leer datos desde teclado

Los programas se hacen para tener interacción con el usuario. Una de las formas de interacción es solicitar la entrada de datos por teclado. Como muchos otros lenguajes de programación, Python también nos ofrece la posibilidad de leer la información introducida por teclado. Para ello se utiliza la función `input()`:

```
>>> name = input('Introduzca su nombre: ')
Introduzca su nombre: Sergio
>>> name
'Sergio'
>>> type(name)
str

>>> age = input('Introduzca su edad: ')
Introduzca su edad: 41
>>> age
'41'
>>> type(age)
str
```

Nota: La función `input()` siempre nos devuelve un objeto de tipo cadena de texto o `str`. Tenerlo muy en cuenta a la hora de trabajar con números, ya que debemos realizar una *conversión explícita*.

Ejercicio

Escriba un programa en Python que lea por teclado dos números enteros y muestre por pantalla el resultado de realizar las operaciones básicas entre ellos.

Ejemplo

- Valores de entrada 7 y 4.

- Salida esperada:

```
7+4=11
7-4=3
7*4=28
7/4=1.75
```

Consejo: Aproveche todo el potencial que ofrece *print()* para conseguir la salida esperada.

3.3.6 Operaciones con «strings»

Combinar cadenas

Podemos combinar dos o más cadenas de texto utilizando el operador `+`:

```
>>> proverb1 = 'Cuando el río suena'
>>> proverb2 = 'agua lleva'

>>> proverb1 + proverb2
'Cuando el río suenaagua lleva'

>>> proverb1 + ', ' + proverb2  # incluimos una coma
'Cuando el río suena, agua lleva'
```

Repetir cadenas

Podemos repetir dos o más cadenas de texto utilizando el operador `*`:

```
>>> reaction = 'Wow'

>>> reaction * 4
'WowWowWowWow'
```

Obtener un caracter

Los «strings» están **indexados** y cada caracter tiene su propia posición. Para obtener un único caracter dentro de una cadena de texto es necesario especificar su **índice** dentro de corchetes `[...]`.

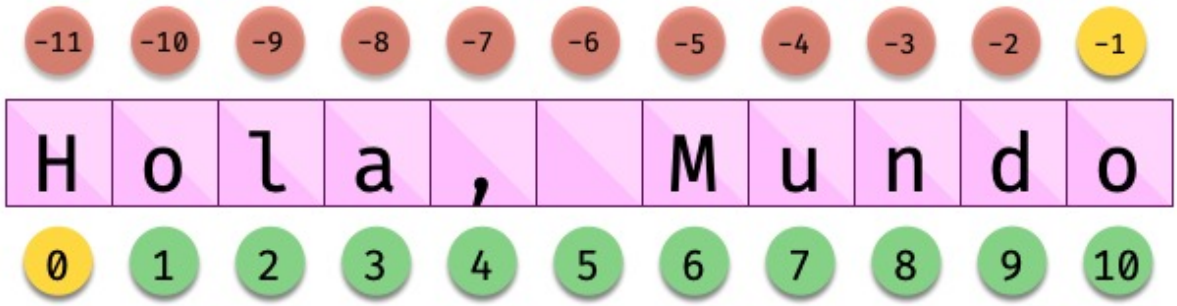

Figura 8: Indexado de una cadena de texto

Veamos algunos ejemplos de acceso a caracteres:

```
>>> sentence = 'Hola, Mundo'

>>> sentence[0]
'H'
>>> sentence[-1]
'o'
>>> sentence[4]
','
>>> sentence[-5]
'M'
```

Truco: Nótese que existen tanto **índices positivos** como **índices negativos** para acceder a cada caracter de la cadena de texto. A priori puede parecer redundante, pero es muy útil en determinados casos.

En caso de que intentemos acceder a un índice que no existe, obtendremos un error por *fuera de rango*:

```
>>> sentence[50]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Advertencia: Téngase en cuenta que el indexado de una cadena de texto siempre empieza en **0** y termina en **una unidad menos de la longitud** de la cadena.

Las cadenas de texto son tipos de datos *inmutables*. Es por ello que no podemos modificar un caracter directamente:

```
>>> song = 'Hey Jude'

>>> song[4] = 'D'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Truco: Existen formas de modificar una cadena de texto que veremos más adelante, aunque realmente no estemos transformando el original sino creando un nuevo objeto con las modificaciones.

Trocear una cadena

Es posible extraer «trozos» («rebanadas») de una cadena de texto[2]. Tenemos varias aproximaciones para ello:

`[:]` Extrae la secuencia entera desde el comienzo hasta el final. Es una especia de **copia** de toda la cadena de texto.

`[start:]` Extrae desde `start` hasta el final de la cadena.

`[:end]` Extrae desde el comienzo de la cadena hasta `end` *menos 1*.

`[start:end]` Extrae desde `start` hasta `end` *menos 1*.

`[start:end:step]` Extrae desde `start` hasta `end` *menos 1* haciendo saltos de tamaño `step`.

Veamos la aplicación de cada uno de estos accesos a través de un ejemplo:

```
>>> proverb = 'Agua pasada no mueve molino'

>>> proverb[:]
'Agua pasada no mueve molino'

>>> proverb[12:]
'no mueve molino'
```

(continué en la próxima página)

[2] El término usado en inglés es *slice*.

(proviene de la página anterior)

```
>>> proverb[:11]
'Agua pasada'

>>> proverb[5:11]
'pasada'

>>> proverb[5:11:2]
'psd'
```

Importante: El troceado siempre llega a una unidad menos del índice final que hayamos especificado. Sin embargo el comienzo sí coincide con el que hemos puesto.

Longitud de una cadena

Para obtener la longitud de una cadena podemos hacer uso de `len()`, una función común a prácticamente todos los tipos y estructuras de datos en Python:

```
>>> proberb = 'Lo cortés no quita lo valiente'
>>> len(proverb)
27

>>> empty = ''
>>> len(empty)
0
```

Pertenencia de un elemento

Si queremos comprobar que una determinada subcadena se encuentra en una cadena de texto utilizamos el operador `in` para ello. Se trata de una expresión que tiene como resultado un valor «booleano» verdadero o falso:

```
>>> proverb = 'Más vale malo conocido que bueno por conocer'

>>> 'malo' in proverb
True

>>> 'bueno' in proverb
True
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> 'regular' in proverb
False
```

Habría que prestar atención al caso en el que intentamos descubrir si una subcadena **no está** en la cadena de texto:

```
>>> dna_sequence = 'ATGAAATTGAAATGGGA'

>>> not('C' in dna_sequence)  # Primera aproximación
True

>>> 'C' not in dna_sequence  # Forma pitónica
True
```

Dividir una cadena

Una tarea muy común al trabajar con cadenas de texto es dividirlas por algún tipo de *separador*. En este sentido, Python nos ofrece la función `split()`, que debemos usar anteponiendo el «string» que queramos dividir:

```
>>> proverb = 'No hay mal que por bien no venga'
>>> proverb.split()
['No', 'hay', 'mal', 'que', 'por', 'bien', 'no', 'venga']

>>> tools = 'Martillo,Sierra,Destornillador'
>>> tools.split(',')
['Martillo', 'Sierra', 'Destornillador']
```

Nota: Si no se especifica un separador, `split()` usa por defecto cualquier secuencia de espacios en blanco, tabuladores y saltos de línea.

Aunque aún no lo hemos visto, lo que devuelve `split()` es una *lista* (otro tipo de datos en Python) donde cada elemento es una parte de la cadena de texto original:

```
>>> game = 'piedra-papel-tijera'

>>> type(game.split('-'))
list
```

Ejercicio

Sabiendo que la longitud de una lista se calcula igual que la *longitud de una cadena de texto*, obtenga el número de palabras que contiene la siguiente cadena de texto:

```
quote = 'Before software can be reusable, it first has to be usable'
```

Existe una forma algo más avanzada de dividir una cadena a través del **particionado**. Para ello podemos valernos de la función `partition()` que proporciona Python.

Esta función toma un argumento como separador, y divide la cadena de texto en 3 partes: lo que queda a la izquiera del separador, el separador en sí mismo y lo que queda a la derecha del separador:

```
>>> text = '3 + 4'

>>> text.partition('+')
('3 ', '+', ' 4')
```

Limpiar cadenas

Cuando leemos datos del usuario o de cualquier fuente externa de información, es bastante probable que se incluyan en esas cadenas de texto, *caracteres de relleno*[3] al comienzo y al final. Python nos ofrece la posibilidad de eliminar estos caracteres u otros que no nos interesen.

La función `strip()` se utiliza para eliminar caracteres del principio y del final de un «string». También existen variantes de esta función para aplicarla únicamente al comienzo o únicamente al final de la cadena de texto.

Supongamos que debemos procesar un fichero con números de serie de un determinado artículo. Cada línea contiene el valor que nos interesa pero se han «colado» ciertos caracteres de relleno que debemos limpiar:

```
>>> serial_number = '\n\t   \n 48374983274832    \n\n\t   \t   \n'

>>> serial_number.strip()
'48374983274832'
```

Nota: Si no se especifican los caracteres a eliminar, `strip()` usa por defecto cualquier combinación de *espacios en blanco*, *saltos de línea* `\n` y *tabuladores* `\t`.

A continuación vamos a hacer «limpieza» por la izquierda (*comienzo*) y por la derecha (*final*) utilizando la función `lstrip()` y `rstrip()` respectivamente:

[3] Se suele utilizar el término inglés «padding» para referirse a estos caracteres.

Lista 6: «Left strip»

```
>>> serial_number.lstrip()
'48374983274832    \n\n\t   \t   \n'
```

Lista 7: «Right strip»

```
>>> serial_number.rstrip()
'\n\t   \n 48374983274832'
```

Como habíamos comentado, también existe la posibilidad de especificar los caracteres que queremos borrar:

```
>>> serial_number.strip('\n')
'\t   \n 48374983274832    \n\n\t   \t   '
```

Importante: La función `strip()` no modifica la cadena que estamos usando (*algo obvio porque los «strings» son inmutables*) sino que devuelve una nueva cadena de texto con las modificaciones pertinentes.

Realizar búsquedas

Aunque hemos visto que la forma pitónica de saber si una subcadena se encuentra dentro de otra es *a través del operador in*, Python nos ofrece distintas alternativas para realizar búsquedas en cadenas de texto.

Vamos a partir de una variable que contiene un trozo de la canción Mediterráneo de *Joan Manuel Serrat* para ejemplificar las distintas opciones que tenemos:

```
>>> lyrics = '''Quizás porque mi niñez
... Sigue jugando en tu playa
... Y escondido tras las cañas
... Duerme mi primer amor
... Llevo tu luz y tu olor
... Por dondequiera que vaya'''
```

Comprobar si una cadena de texto **empieza o termina por alguna subcadena**:

```
>>> lyrics.startswith('Quizás')
True

>>> lyrics.endswith('Final')
False
```

Encontrar la **primera ocurrencia** de alguna subcadena:

```
>>> lyrics.find('amor')
93

>>> lyrics.index('amor')  # Same behaviour?
93
```

Tanto `find()` como `index()` devuelven el **índice** de la primera ocurrencia de la subcadena que estemos buscando, pero se diferencian en su comportamiento cuando la subcadena buscada no existe:

```
>>> lyrics.find('universo')
-1

>>> lyrics.index('universo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Contabilizar el **número de veces que aparece** una subcadena:

```
>>> lyrics.count('mi')
2

>>> lyrics.count('tu')
3

>>> lyrics.count('él')
0
```

Ejercicio

Dada la siguiente letra[5], obtenga la misma pero sustituyendo la palabra `voices` por `sounds`:

```
>>> song = '''You look so beautiful in this light
... Your silhouette over me
... The way it brings out the blue in your eyes
... Is the Tenerife sea
... And all of the voices surrounding us here
... They just fade out when you take a breath
... Just say the word and I will disappear
... Into the wilderness'''
```

Utilice para ello únicamente búsqueda, concatenación y troceado de cadenas de texto.

[5] «Tenerife Sea» por Ed Sheeran.

Reemplazar elementos

Podemos usar la función `replace()` indicando la *subcadena a reemplazar*, la *subcadena de reemplazo* y *cuántas instancias* se deben reemplazar. Si no se especifica este último argumento, la sustitución se hará en todas las instancias encontradas:

```
>>> proverb = 'Quien mal anda mal acaba'

>>> proverb.replace('mal', 'bien')
'Quien bien anda bien acaba'

>>> proverb.replace('mal', 'bien', 1)  # sólo 1 reemplazo
'Quien bien anda mal acaba'
```

Mayúsculas y minúsculas

Python nos permite realizar variaciones en los caracteres de una cadena de texto para pasarlos a mayúsculas y/o minúsculas. Veamos las distintas opciones disponibles:

```
>>> proverb = 'quien a buen árbol se arrima Buena Sombra le cobija'

>>> proverb
'quien a buen árbol se arrima Buena Sombra le cobija'

>>> proverb.capitalize()
'Quien a buen árbol se arrima buena sombra le cobija'

>>> proverb.title()
'Quien A Buen Árbol Se Arrima Buena Sombra Le Cobija'

>>> proverb.upper()
'QUIEN A BUEN ÁRBOL SE ARRIMA BUENA SOMBRA LE COBIJA'

>>> proverb.lower()
'quien a buen árbol se arrima buena sombra le cobija'

>>> proverb.swapcase()
'QUIEN A BUEN ÁRBOL SE ARRIMA bUENA sOMBRA LE COBIJA'
```

Identificando caracteres

Hay veces que recibimos información textual de distintas fuentes de las que necesitamos identificar qué tipo de caracteres contienen. Para ello Python nos ofrece un grupo de funciones.

Veamos **algunas** de estas funciones:

Lista 8: Detectar si todos los caracteres son letras o números

```
>>> 'R2D2'.isalnum()
True
>>> 'C3-PO'.isalnum()
False
```

Lista 9: Detectar si todos los caracteres son números

```
>>> '314'.isnumeric()
True
>>> '3.14'.isnumeric()
False
```

Lista 10: Detectar si todos los caracteres son letras

```
>>> 'abc'.isalpha()
True
>>> 'a-b-c'.isalpha()
False
```

Lista 11: Detectar mayúsculas/minúsculas

```
>>> 'BIG'.isupper()
True
>>> 'small'.islower()
True
>>> 'First Heading'.istitle()
True
```

3.3.7 Interpolación de cadenas

En este apartado veremos cómo **interpolar** valores dentro de cadenas de texto utilizando diferentes formatos. Interpolar (en este contexto) significa sustituir una variable por su valor dentro de una cadena de texto.

Veamos los estilos que proporciona Python para este cometido:

Nombre	Símbolo	Soportado
Estilo antiguo	`%`	>= Python2
Estilo «nuevo»	`.format`	>= Python2.6
«f-strings»	`f''`	>= Python3.6

Aunque aún podemos encontrar código con el estilo antiguo y el estilo nuevo en el formateo de cadenas, vamos a centrarnos en el análisis de los **«f-strings»** que se están utilizando bastante en la actualidad.

«f-strings»

Los **f-strings** aparecieron en Python 3.6 y se suelen usar en código de nueva creación. Es la forma más potente – y en muchas ocasiones más eficiente – de formar cadenas de texto incluyendo valores de otras variables.

La **interpolación** en cadenas de texto es un concepto que existe en la gran mayoría de lenguajes de programación y hace referencia al hecho de sustituir los nombres de variables por sus valores cuando se construye un «string».

Para indicar en Python que una cadena es un «f-string» basta con precederla de una `f` e incluir las variables o expresiones a interpolar entre llaves `{...}`.

Supongamos que disponemos de los datos de una persona y queremos formar una frase de bienvenida con ellos:

```
>>> name = 'Elon Musk'
>>> age = 49
>>> fortune = 43_300

>>> f'Me llamo {name}, tengo {age} años y una fortuna de {fortune} millones'
'Me llamo Elon Musk, tengo 49 años y una fortuna de 43300 millones'
```

Advertencia: Si olvidamos poner la **f** delante del «string» no conseguiremos sustitución de variables.

Podría surgir la duda de cómo incluir llaves dentro de la cadena de texto, teniendo en cuenta que las llaves son símbolos especiales para la interpolación de variables. La respuesta es duplicar las llaves:

```
>>> x = 10

>>> f'The variable is {{ x = {x} }}'
'The variable is { x = 10 }'
```

Formateando cadenas

Nivel intermedio

Los «f-strings» proporcionan una gran variedad de **opciones de formateado**: ancho del texto, número de decimales, tamaño de la cifra, alineación, etc. Muchas de estas facilidades se pueden consultar en el artículo Best of Python3.6 f-strings[4]

Dando formato a valores enteros:

```
>>> mount_height = 3718

>>> f'{mount_height:10d}'
'      3718'

>>> f'{mount_height:010d}'
'0000003718'
```

Dando formato a otras bases:

```
>>> value = 0b10010011
```

(continué en la próxima página)

[4] Escrito por Nirant Kasliwal en Medium.

(proviene de la página anterior)

```
>>> f'{value}'
'147'
>>> f'{value:b}'
'10010011'

>>> value = 0o47622
>>> f'{value}'
'20370'
>>> f'{value:o}'
'47622'

>>> value = 0xab217
>>> f'{value}'
'700951'
>>> f'{value:x}'
'ab217'
```

Dando formato a valores flotantes:

```
>>> pi = 3.14159265

>>> f'{pi:f}'  # 6 decimales por defecto (se rellenan con ceros si procede)
'3.141593'

>>> f'{pi:.3f}'
'3.142'

>>> f'{pi:12f}'
'    3.141593'

>>> f'{pi:7.2f}'
'   3.14'

>>> f'{pi:07.2f}'
'0003.14'

>>> f'{pi:.010f}'
'3.1415926500'

>>> f'{pi:e}'
'3.141593e+00'
```

Alineando valores:

```
>>> text1 = 'how'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> text2 = 'are'
>>> text3 = 'you'

>>> f'{text1:<7s}|{text2:^11s}|{text3:>7s}'
'how    |    are    |    you'

>>> f'{text1:-<7s}|{text2:·^11s}|{text3:->7s}'
'how----|····are····|----you'
```

Modo «debug»

A partir de Python 3.8, los «f-strings» permiten imprimir el nombre de la variable y su valor, como un atajo para depurar nuestro código. Para ello sólo tenemos que incluir un símbolo = después del nombre de la variable:

```
>>> serie = 'The Simpsons'
>>> imdb_rating = 8.7
>>> num_seasons = 30

>>> f'{serie=}'
"serie='The Simpsons'"

>>> f'{imdb_rating=}'
'imdb_rating=8.7'

>>> f'{serie[4:]=}'  # incluso podemos añadir expresiones!
"serie[4:]='Simpsons'"

>>> f'{imdb_rating / num_seasons=}'
'imdb_rating / num_seasons=0.29'
```

Ejercicio

Dada la variable:

```
e = 2.71828
```

, obtenga los siguientes resultados utilizando «f-strings»:

```
'2.718'
'2.718280'
'    2.72'  # 4 espacios en blanco
'2.718280e+00'
```

(continué en la próxima página)

(proviene de la página anterior)

```
'00002.7183'
'            2.71828'  # 12 espacios en blanco
```

3.3.8 Caracteres Unicode

Python trabaja *por defecto* con caracteres **Unicode**. Eso significa que tenemos acceso a la amplia carta de caracteres que nos ofrece este estándar de codificación.

Supongamos un ejemplo sobre el típico «emoji» de un **cohete** definido en este cuadro:

Figura 9: Representación Unicode del carácter ROCKET

La función `chr()` permite representar un carácter **a partir de su código**:

```
>>> rocket_code = 0x1F680
>>> rocket = chr(rocket_code)
>>> rocket
'🚀'
```

La función `ord()` permite obtener el código (decimal) de un carácter **a partir de su representación**:

```
>>> rocket_code = hex(ord(rocket))
>>> rocket_code
'0x1f680'
```

El modificador `\N` permite representar un carácter **a partir de su nombre**:

```
>>> '\N{ROCKET}'
'🚀'
```

3.3.9 Casos de uso

Nivel avanzado

Hemos estado usando muchas funciones de objetos tipo «string» (y de otros tipos previamente). Pero quizás no sabemos aún como podemos descubrir todo lo que podemos hacer con ellos y los **casos de uso** que nos ofrece.

Python proporciona una *función «built-in»* llamada `dir()` para inspeccionar un determinado tipo de objeto:

```
>>> text = 'This is it!'

>>> dir(text)
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
```

(continué en la próxima página)

(proviene de la página anterior)

```
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

Esto es aplicable tanto a variables como a literales e incluso a tipos de datos (clases) explícitos:

```
>>> dir(10)
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 ...
 'imag',
 'numerator',
 'real',
 'to_bytes']

>>> dir(float)
['__abs__',
 '__add__',
 '__bool__',
 '__class__',
 ...
 'hex',
 'imag',
 'is_integer',
 'real']
```

EJERCICIOS DE REPASO

1. Escriba un programa en Python que acepte el nombre y los apellidos de una persona y los imprima en orden inverso separados por una coma. Utilice *f-strings* para implementarlo (`solución`).

   Entrada: nombre=Sergio; apellidos=Delgado Quintero
   Salida: Delgado Quintero, Sergio

2. Escriba un programa en Python que acepte una ruta remota de recurso samba, y lo separe en nombre(IP) del equipo y ruta (`solución`).

   Entrada: //1.1.1.1/eoi/python
   Salida: equipo=1.1.1.1; ruta=/eoi/python

3. Escriba un programa en Python que acepte un «string» con los 8 dígitos de un NIF, y calcule su dígito de control (`solución`).

   Entrada: 12345678
   Salida: 12345678Z

4. Escriba un programa en Python que acepte un entero `n` y compute el valor de `n + nn + nnn` (`solución`).

   Entrada: 5
   Salida: 615

5. Escriba un programa en Python que acepte una palabra en castellano y calcule una métrica que sea el número total de caracteres de la palabra multiplicado por el número total de vocales que contiene la palabra (`solución`).

   Entrada: ordenador
   Salida: 36

AMPLIAR CONOCIMIENTOS

- A Guide to the Newer Python String Format Techniques
- Strings and Character Data in Python
- How to Convert a Python String to int
- Your Guide to the Python print<> Function
- Basic Input, Output, and String Formatting in Python
- Unicode & Character Encodings in Python: A Painless Guide
- Python String Formatting Tips & Best Practices
- Python 3's f-Strings: An Improved String Formatting Syntax
- Splitting, Concatenating, and Joining Strings in Python
- Conditional Statements in Python
- Python String Formatting Best Practices