# Teoría FP1:

## Entrada de datos

Un algoritmo describe la solución de un problema en función de los datos necesarios para representar un caso concreto del problema y de los pasos necesarios para obtener el resultado deseado. El tratamiento de la información plantea problemas cuyo propósito es hallar un resultado desconocido a partir de datos conocidos.

Para que un ordenador resuelva un caso de un problema, teniendo el programa para hacerlo, es necesario proporcionarle los datos del caso; esto se puede hacer mediante instrucciones de entrada. Véase el siguiente programa:

```
print("Hola ¿Cómo te llamas?")
name = input()
print("Hola", name)
```

El resultado de la ejecución de este programa, suponiendo que el usuario introdujera por teclado el nombre Pedro, sería el siguiente:

```
Hola ¿Cómo te llamas? Pedro
Hola Pedro
```

El programa usa dos veces la instrucción print, que es una **instrucción de salida**, es decir, una instrucción para dar información, y aparece una instrucción nueva, input, que es una **instrucción de entrada**, una instrucción para recoger datos que van a ser usados por el programa.

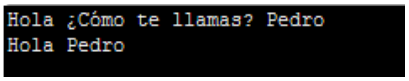
También aparece otro elemento, *name*, que no es una instrucción, sino el nombre de una **variable**. Una variable se puede ver como una etiqueta que se usa para hacer referencia a un dato manejado por el programa.

El valor del dato referenciado por una variable puede ser distinto en diferentes ejecuciones del programa (supóngase que el programa anterior se ejecuta para otro usuario llamado Juan) y puede, incluso, variar durante la ejecución del programa (de ahí el nombre de variable). Esta capacidad de variación de los datos es lo que permite que un programa resuelva diferentes casos de un mismo problema.

El efecto de la ejecución de la instrucción de entrada input es que el programa espera a que el usuario teclee el valor del dato deseado, terminando con la tecla ↵ (denominada intro o entrar, en inglés *enter*). Una vez que lo hace, se recoge ese valor y, tal como indica el código mostrado más arriba, se asigna a la variable que está a la izquierda del símbolo de **asignación =**. Como el programa se va a detener hasta que el usuario introduzca el valor de un dato, lo normal es que se le indique justo antes con un mensaje qué dato es el que se espera que introduzca; esto se hace en el ejemplo anterior con la primera instrucción print, pero se puede abreviar usando la propia instrucción input:

```
name = input("Hola ¿Cómo te llamas? ")
print("Hola", name)
```

El resultado de la ejecución de este programa es:




Nótese que en este caso la entrada del valor se espera en la misma línea donde se muestra el mensaje que lo solicita, de ahí que en la instrucción input hayamos añadido un espacio después del '?', para evitar que el nombre que teclee el usuario se visualice pegado al '?' (si el usuario tecleara un espacio antes del nombre, ese espacio formaría parte del contenido de la variable name, cosa que en principio no queremos).

En ambas versiones del programa, la última instrucción muestra un saludo personalizado que incluye el valor del dato introducido por el usuario:

```
print("Hola", name)
```

**name** no se encierra entre comillas; las comillas sirven para indicar un valor literal tal cual (la palabra Hola en el ejemplo). En el caso de la variable name, lo que se quiere mostrar no es la palabra name, sino el dato referencia do por la variable (Pedro en el ejemplo).

### Los nombres de las variables

Los programas usan **variables** para hacer referencia a los datos del problema que resuelven: datos de inicio, que varían de un caso del problema a otro; datos intermedios, generados durante la resolución del problema, y que pueden variar a medida que se ejecuta el programa; y datos finales, con los resultados que conforman la solución del problema.

Las variables tienen un **nombre** y hacen referencia al **valor** de un dato que se almacena en la memoria del ordenador accesible a la actual ejecución del programa.

Las **reglas léxicas** de Python requieren que los nombres de las variables empiecen por una letra mayúscula (A··Z), o minúscula (a··z)), o un guion bajo/subrayado (_), pudiendo continuar con cualquier combinación de letras, guiones bajos y dígitos (0··9):

```
nombre
NOMBRE
_nombre_1
fecha_de_nacimiento
xASFasDasd24_d
```

Las **reglas de estilo** de Python establecen que los nombres de las variables deben escribirse en minúsculas, separando por guiones bajos sus componentes cuando se trate de nombres compuestos, y ser descriptivos respecto al dato que representan:

```
nombre
NOMBRE
_nombre_1
```

```
fecha_de_nacimiento
xASFasDasd24_d
```

## Los valores de las variables

Los programas manejan datos usando variables. Las variables tienen un **nombre** (**identificador**) y hacen referencia (contienen) al **valor** de un dato. Para asociar (**asignar**) un valor a una variable, se usa una **instrucción de asignación**, caracterizada por el símbolo =.

```
name = input()          # asigna a name unvalor introducido por el usuario
greeting = "Hola, mundo" # asigna a greeting un texto literal
```

Podemos definir un **dato** como "una representación simbólica (numérica, alfabética, algorítmica, espacial, etc.) de un atributo cualitativo o una magnitud cuantitativa". Existen muchos tipos de datos diferentes. En el ejemplo anterior, a las variables name y greeting se les asignan valores que representan un pequeño texto. Los valores que representan texto se conocen como **strings**. Una string es simplemente una **secuencia de caracteres**. Las strings pueden usarse para representar cosas como nombres, notas, observaciones, etc.

Otros posibles tipos de datos que se pueden asociar una variable son los números; por ejemplo, números **enteros** (sin parte fraccionaria):

```
age = 21
```

números **reales** (que incluyen parte fraccionaria, antes de la cual se ha de usar el punto, no la coma):

```
temperature = 36.5
```

números **complejos** (con parte real y parte imaginaria; la unidad imaginaria se representa por **j**):

```
complex_number = -34+5.3j
```

También se pueden usar valores **booleanos**, que se representan con **True** (verdadero) o **False** (falso):

```
bool_data = True
```

Los valores booleanos pueden ser el resultado de una comparación u otro tipo de operación:

```
bool_data = 8 > 5 # se asigna elvalor False, resultado de laexpresión 8 > 5
```

Nótese que solo los **literales** de valores de tipo texto (strings) se representan encerrados entre comillas.

Las strings y los números (incluyendo los booleanos) son tipos **básicos** en Python. Python predefine también otros tipos de datos (tuplas, listas, diccionarios, ...) o los incorpora a través de librerías externas, aparte de dar la posibilidad al programador de definir nuevos tipos según sus necesidades.

## Asignación

Los programas manejan datos usando variables. Las variables tienen un **nombre** y hacen referencia al **valor** de un dato. Para asociar un (nuevo) valor con una variable, se usa una **instrucción de asignación**, caracterizada por el símbolo **=**. A la izquierda del **=** se indica una variable y a su derecha un valor (literal o a obtener de una cálculo u operación).

En la **ejecución** de una instrucción de asignación, **una vez que se obtiene el valor resultante del lado derecho** del **=**, se le asigna a continuación a la variable del lado izquierdo, sustituyendo al valor que contuviera previamente.

A una variable se le puede asignar:

- El resultado de una operación de entrada

```
var1 = input()
```

- Un valor expresado literalmente (recuérdese que solo las strings literales se encierran entre comillas)

```
var2 = 35
var3 = "Hola"
```

- Un cálculo (se asigna el valor resultante)

```
var4 = 7 * 5 + 18 # a var4 se le asigna elvalor 53
```

- Otra variable (para tipos básicos de Python, a los efectos, se asigna una copia del valor de la variable de la derecha)

```
var5 = var4
```

- Las expresiones pueden incluir variables

```
var6 = var4 + var5 - 10 # elvalor asignado a var6 es el resultado 96
```

- La propia variable a la que se asigna el resultado de una expresión puede estar incluida en la expresión (el mismo nombre de variable puede aparecer a ambos lados del **=**, pero

significa cosas distintas: en el lado derecho se toma su  para realizar un cálculo, mientras que en el lado izquierdo indica el  al que vamos a asignar ese valor)

valor

destino

```
var6 = var6 + 1 # elvalor asignado a var6 es 97
```

- Si la variable forma parte de la expresión, se puede abreviar en algunos casos frecuentes

```
var6 += 1 # equivale a var6 = var6 + 1
var6 *= 2 # equivale a var6 = var6 * 2
```

Una vez vistos los ejemplos, podemos resumir diciendo que una instrucción de asignación se compone siempre de un símbolo = con una **expresión** a su derecha (que puede ser muy simple, como un literal o una variable, o contener operaciones), cuyo valor resultante se asigna a la variable de la izquierda.

## Asignación múltiple

Los programas manejan datos usando variables. Las variables tienen un **nombre** y hacen referencia al **valor** de un dato. Para asociar un valor con una variable, se usa una **instrucción de asignación,** representada por el símbolo =. El funcionamiento de la asignación es: primero se evalúa la expresión que está a la derecha del símbolo de asignación y, luego, el resultado de esa evaluación se asigna a la variable que está a la izquierda del símbolo de asignación.

```
var1 = 35
```

Además de lo dicho, Python permite asignar múltiples valores a múltiples variables en una sola instrucción de asignación:

```
a, b, c = 1, 2, 3
```

A la derecha de la asignación debe haber tantas expresiones, separadas por comas, como variables haya a la izquierda de la asignación. Los valores de las expresiones se asignan en paralelo ("simultáneamente") de izquierda a derecha a las variables correspondientes: el primer valor a la primera variable, el segundo valor a la segunda variable, etc. En el ejemplo anterior, el valor 1 se asigna a la variable a, el valor 2 a la variable b y el valor 3 a la variable c.

La asignación múltiple permite, entre otras cosas, intercambiar cómodamente los valores de dos variables con una sola instrucción (en el ejemplo a recibe el anterior valor de b, y simultáneamente, b recibe el anterior valor de a).

```
a, b = b, a
```

Esto mismo, en muchos otros lenguajes, requeriría tres instrucciones y una variable adicional:

```
aux = a
a = b
b = aux
```

La variable adicional (aux en el ejemplo) sirve para guardar el valor de la primera variable que se va a cambiar, de forma que no se pierda y pueda asignársele luego a la otra.

## Entrada de valores no textuales

La instrucción input recoge los datos, sean del tipo que sean, como una secuencia de caracteres (string), tal como se introducen por teclado.

```
name = input("¿Cómo te llamas? ")
```

Si la secuencia de caracteres introducidos se puede interpretar como un valor válido de otro tipo primitivo (número entero, número real, número complejo), hay que indicar que se quiere hacer esa **conversión** (el caso bool es especial):

```
text = input("Dame unastring formada por una secuencia dedígitos: ")
integer_number = int(input("Dame un número entero: "))
real_number = float(input("Dame un número real: "))
complex_number = complex(input("Dame un número complejo: "))
bool_value = bool(int(input("Dame unbooleano (0→False, 1→True): ")))
```

Resultado de la ejecución (se resaltan los valores introducidos por el usuario):

```
Dame una string formada por una secuencia de dígitos: 23289 Dame un número entero: 23289 Dame un número real:
23289.0 Dame un número complejo: 14+23j Dame un booleano (0→False, 1→True): 0
```

Nótese el uso de **int**, **float** o **complex** para indicar la interpretación requerida. Las strings (secuencias de caracteres) se identifican como **str**, pero sería redundante indicarlo en una instrucción de entrada, dado que ese es el tipo por omisión de los datos entrados por teclado. El caso de **bool()** es especial como veremos, no nos serviría teclear False para indicar este valor, de ahí que recurramos a los int 0 y 1 para luego convertir a bool.

En caso de que la secuencia de caracteres tecleada por el usuario no pueda interpretarse como un valor del tipo requerido, se produce un error.

Cada tipo de datos usa una representación interna en memoria diferente como secuencia de bits (ceros y unos), en cada caso usando una codificación diferente. Incluso en casos como los referenciados por las variables text e integer_number en el ejemplo, que responden a la misma secuencia de caracteres de entrada: en el primer caso cada caracter (de "cifra") se representa en binario por separado de acuerdo con su codificación estándar (p.e. cada '2' se podría representar por una secuencia 00110010), mientras que en el segundo se convierte a un valor numérico binario equivalente (en notación denominada *complemento a dos*), concretamente 0··0101101011111001 (23289 en binario).

El nombre del tipo *float*, que usamos para los números reales, se debe a que usa una representación interna llamada *floating point* (coma flotante), análoga a la notación científica de nuestras calculadoras (que usa un exponente), de forma que la coma (el punto) no está siempre justo a la izquierda de la parte fraccionaria (*fixed point*), sino que "flota" en combinación con el exponente, p.e. 123.45 (escrito en coma fija) podemos imaginar que internamente se representa como 0.12345·103 (en realidad, algo similar pero adaptado a binario), o sea, el punto ha "flotado" hasta justo antes de la primera cifra distinta de cero adecuándose al mismo tiempo el exponente. De esta manera, el número queda representado por un par de enteros, en el ejemplo 12345 (la *mantisa*) y 3 (el exponente). El tamaño de la mantisa (y del exponente) es limitado, por lo que en muchos casos solo podemos representar un número real de forma aproximada.

# Tipos de datos

Los programas manejan datos usando variables. Wikipedia define un **dato** como "una representación simbólica (numérica, alfabética, algorítmica, espacial, etc.) de un atributo o variable cuantitativa o cualitativa".

Existen datos de diferentes tipos: texto, como el nombre de una persona, números, como la edad de una persona, fechas, como la fecha de nacimiento de una persona, etc.

Los tipos de datos estándar de Python incluyen números (enteros, reales, complejos y booleanos), strings (texto), tuplas, listas y diccionarios. Las strings, las tuplas y las listas se agrupan en la categoría de secuencias.

Por ahora, nos centraremos en los números y las strings.

## Números

Los números en Python pueden ser: enteros (int), reales (float) y complejos (complex).

Para los literales de int, un 0o o 0O antes de las cifras indica que el resto de la secuencia representa un número en base 8 (octal), en lugar de la habitual base 10 (decimal), y el prefijo 0x o 0X indica que está en base 16 (hexadecimal); en cualquier caso, el valor numérico se representará internamente por el número en base 2 que corresponda (p.e. el mismo para 10, 0o12 y 0xa, ya que todos ellos representan el valor entero diez). El punto o/y la e/E (que se lee "por diez elevado a") hace que un literal numérico represente un valor float. La unidad imaginaria j/J, que tiene que venir precedida de un número (incluso 0), hace que sea de complex

Todos los tipos numéricos admiten las siguientes operaciones (aunque por simplicidad se han usado enteros en los ejemplos):

| **Operación** | **Resultado** |
|---|---|
| x **+** y | Suma de x más y: 5 + 2 da 7 |
| x **-** y | Resta de x menos y: 5 - 2 da 3 |
| x ***** y | Producto de x por y: 5 * 2 da 10 |

| | |
|---|---|
| x **/** y | Cociente de dividir x entre y: 5 / 2 da 2.5 |
| x **//** y | Cociente entero de x entre y: 5 // 2 da 2*El número de veces que el divisor cabe entero en el dividendo: 3.14 // 0.7 da 4.0(Si el dividendo o el divisor son float, el resultado es float, si ambos son int, el resultado es int)* |
| x **%** y | Resto de x // y: 5 % 2 da 1; 3.14 % 0.7 da 0.34*Se mantiene el signo del divisor : 5 % -2 da -1* |
| **-**x | x negada: 5 si x vale -5 |
| **+**x | x sin cambio: -5 si x vale -5 |
| **abs**(x) | Valor absoluto de x: abs(-5) da 5 |
| **divmod**(x, y) | La pareja (x // y, x % y): divmod(5, 2) da (2, 1) |
| **pow**(x, y) | x elevado a y: pow(5, 2) da 25 |
| x ****** y | x elevado a y: 5 ** 2 da 25 |

## Valores booleanos

El tipo bool solo dispone de dos valores, que representamos por los literales *True* y *False*. En Python se le considera un subtipo de int, y los siguientes valores numéricos se toman como *False* al emplearse donde se espera normalmente un bool (en caso contrario, se tomaría el valor *True*):

- El valor cero de cualquier tipo numérico: 0 (int), 0.0 (float), 0j (complex)
- Una secuencia vacía, p.e. la string "", compuesta de cero caracteres (aunque no por ello deja de ser una string)
- El valor que representamos por el literal *None*, un valor especial que, aunque significa "ningún valor", ¡es un valor!, de hecho es el único valor del tipo *NoneType*

En general, a cualquier

valor

x del tipo que sea podemos aplicarle bool(x), que nos dará un

valor

bool consistente con lo dicho. Naturalmente, bool(False) da False y bool(True) da True.

Los operadores **or**, **and** y **not** esperan normalmente operandos de tipo bool:

| **Operación** | **Resultado** |
|---|---|
| x **or** y | Si ambos son bool, el resultado es *False* solo si **los dos** son *False*.<br>Las<br>operandos se evalúan de izquierda a derecha, de manera que si bool(x) es *True*, el resultado es el valor de x, no evaluándose y; en caso contrario el resultado será el valor de y |

| | |
|---|---|
| x **and** y | Si ambos son bool, el resultado es *True* solo si **los dos** son *True*.<br>Las<br>operandos se evalúan de izquierda a derecha, de manera que si bool(x) es *False*, el resultado es el valor de x, no evaluándose y; en caso contrario el resultado será el valor de y |
| **not** x | Negación: si bool(x) da *False*, el resultado es *True*, si no, es *False* |

Por el funcionamiento de or y and aquí descrito se dice que son operadores **cortocircuito**, porque a diferencia de lo usual el valor de un operando puede "cortocircuitar" (evitar la evaluación) de otro(s) operando(s); lo cual con cierta frecuencia es útil, en particular si la evaluación cortocircuitada en ese caso produciría un error.

## Strings

Las strings representan secuencias contiguas de caracteres. Los literales de valores de tipo string (str), se encierran entre comillas dobles o simples:

```
var1 = "hola"
var2 = 'hola'
```

```
str_var = "Hello world"
print(str_var)        # Muestra "Hello world"
```

La longitud de una string (número de caracteres) se obtiene programáticamente usando la función

len():

```
str_var = "Hello world"
print(len(str_var))    # Muestra elvalor entero 11
```

Se puede acceder a un carácter concreto de una string escribiendo la string, o el nombre de la variable que la referencia, seguida del índice de la posición que ocupa el carácter deseado entre corchetes. El índice de la primera posición es 0.

```
str_var = "Hello world"
print(str_var[0])      # Muestra el primer carácter de str_var: H
print(str_var[4])      # Muestra el quinto carácter de str_var: o
```

También se pueden usar índices relativos a la longitud de la string:

```
str_var = "Hello world"
print(str_var[-1])     # Muestra el último carácter de str_var: d
print(str_var[-5])     # Muestra el quinto carácter, empezando por el final, de str_var: w
```

Se puede obtener una substring (subsecuencia) de una string usando el operador de segmento o *slice* **[:]**. El segmento deseado se identifica indicando el índice del primer elemento y el índice de la posición siguiente al último elemento. Si se omite el segundo valor , se toma el segmento desde el primer índice hasta el final; si se omite el primero, toma desde el principio hasta la posición del índice anterior al indicado:

```
str_var = "Hello world"
print(str_var[2:5])    # Muestra los caracteres de 3º al 5º: "llo"
print(str_var[2:])     # Muestra los caracteres a partir del 3º: "llo world"
print(str_var[:5])     # Muestra los caracteres hasta el 5º: "Hello"
```

El diccionario de la RAE define **concatenar** como "**unir dos o más cosas**"; en el caso de las strings, concatenar dos strings consiste en formar una nueva string juntando, en orden, dos preexistentes.

En Python existen dos operadores de concatenación: el signo más (+) es el operador de concatenación de strings y el asterisco (*) es el operador de repetición (equivalente a realizar una concatenación de la misma string repetidas veces).

```
str_var = "Hello world"
print('a' + 'a')       # Muestra "aa"
print(str_var + "TEST") # Muestra elvalor de str_var concatenado con "TEST": "Hello world TEST"
print('ab' * 5)        # Muestra "abababababab"
print(str_var * 2)     # Muestra "Hello worldHello world"
```

Podríamos decir que la repetición de strings es a la concatenación de strings lo que la multiplicación de números enteros es a la suma de números enteros. Nótese que la concatenación junta directamente las strings, sin añadir ningún carácter en medio.

## Conocer el tipo de un dato

Podemos conocer el tipo de un dato usando la operación *type()*.

```
a = 200
b = "200"
t1 = type(a)
t2 = type(b)
print(t1)
print(t2)
```

```
<type 'int'>
<type 'str'>
```

Nótese que el valor devuelto por *type()* es un valor de tipo *type*:

```
print(type(t1))
```

```
<type 'type'>
```

El valor, aunque se muestra con el formato de los ejemplos, es uno de los tipos de datos (*int*, *float*, *str*, ...):

```
print(t1 == int)
```

```
True
```

Si queremos obtener una string con solo el nombre del tipo, podemos consultar el atributo ***__name__*** (un atributo es una propiedad asociada a un dato):

```
typeName = t1.__name__
print(typeName)
print(type(typeName))
```

```
int
<type 'str'>
```

## Type casting

En programación se conoce como "type casting" a obtener, a partir de un valor de un tipo T1, un valor "similar" de otro tipo T2; lo que no en todos los casos será posible. Si T1 y T2 son el mismo tipo, es trivial, obtenemos un valor igual de ese tipo.

**int()** construye un número entero a partir de: un número real (*truncándolo*, esto es, eliminando la parte fraccionaria), o una string (siempre que la string represente un número entero)

```
a = int(10)    # a toma elvalor entero 10
b = int(3.7)   # b toma elvalor entero 3
c = int("40")  # c toma elvalor entero 40
s = "25"
d = int(s)     # d toma elvalor entero 25
```

**float()** construye un número real a partir de: un número entero, o una string (siempre que la string represente un número real o un número entero)

```
a = float(10)    # a toma elvalor real 10.0
b = float(3.5)   # b toma elvalor real 3.5
c = float("40")  # c toma elvalor real 40.0
```

```
s = "25.0"
d = float(s)     # d toma elvalor real 25.0
```

**complex()** construye un número complejo a partir de: un número entero, un número real, o una string (siempre que la string represente un número real, un número entero o un número complejo)

```
a = complex(10)    # a toma elvalor complejo 10+0j
b = complex(3.5)   # b toma elvalor complejo 3.5+0j
c = complex("40")  # c toma elvalor complejo 40+0j
s = "25"
d = complex(s)     # d toma elvalor complejo 25+0j
e = complex(1+2j)  # e toma elvalor complejo 1+2j
```

**str()** construye una string a partir de casi cualquier otro tipo de datos.

```
a = str(10)    # a toma elvalorstring "10"
b = str(3.5)   # b toma elvalorstring "3.5"
c = str("40")  # c toma elvalorstring "40"
s = "25"
d = str(s)     # d toma elvalorstring "25"
e = str(1+2j)  # e toma elvalorstring "1+2j"
```

## Anotación de tipos

Python es un lenguaje de programación con *tipado dinámico*, lo que significa que el tipo de una variable (el tipo del dato asociado a la variable) se establece en el momento de asignarle un valor, tipo que cambiará si al asignarle un nuevo valor este fuera de un tipo diferente; a diferencia de los lenguajes con *tipado estático*, en los que se requiere declarar de qué tipo es cada variable, la cual solo podrá contener valores de dicho tipo (lo que podrá requerir casting según el caso; de lo contrario resultará en una situación de error intentar asignarle un valor de tipo distinto). Veamos las siguientes asignaciones en Python:

```
a = 0   # el tipo de a es int
b = 0.0 # el tipo de b es float
c = "0" # el tipo de c es str
a = 0.0 # el tipo de a cambia a float
```

En Python, no obstante, es posible "**anotar**" el tipo de una variable, lo que en principio solo tiene un efecto informativo, equivalente a un comentario, acerca del tipo de valor que se espera que contenga:

```
a: int = 0     # el tipo de a es (y debiera seguir siendo) int
b: float = 0.0 # el tipo de b es (y debiera seguir siendo) float
```

```
c: str = "0"   # el tipo de c es (y debiera seguir siendo) str
```

A diferencia de los lenguajes con tipado estático, en Python a una variable que haya sido anotada con un tipo se le puede asignar más tarde un valor de un tipo diferente; aunque hay herramientas que pueden analizar el código y señalar situaciones que supongan riesgo de que durante la ejecución algún contenido no se ajuste a la intención manifestada en la anotación.

## Operadores de comparación

Estos operadores son aplicables a muchos tipos de datos; comparan los valores de sus dos operandos y devuelven un valor booleano en función de la relación entre ellos, por lo que técnicamente se les llama **operadores relacionales**.

Supongamos, en el siguiente ejemplo, que la variable *a* vale 10 y la variable *b* 20:

| operador | descripción | ejemplo |
|---|---|---|
| == | compara si son iguales | a == b da False |
| != | compara si son distintos | a != b da True |
| > | compara si el primero es mayor que el segundo | a > b da False |
| < | compara si el primero es menor que el segundo | a < b da True |
| >= | compara si el primero es mayor o igual que el segundo | a >= b da False |
| <= | compara si el primero es menor o igual que el segundo | a <= b da True |

## Precedencia de operadores

Una expresión es una combinación de operandos y operadores. En el siguiente ejemplo, + es un operador obivamente de adición, y los números 3 y 8 son sus operandos (también podrían ser variables o cualquier otro elemento del lenguaje que produjera un valor, en este caso aritmético):

```
3 + 8
```

La expresión anterior es una expresión bastante simple, pues solo tiene un operador (podría no tener ninguno y constar de un único "operando"). En una expresión puede haber más de un operador; cuando esto ocurre, las reglas de **precedencia** determinan el orden de aplicación de los distintos operadores. Por ejemplo, la multiplicación (representada por el *) tiene una precedencia mayor que la adición, por tanto, la expresión

```
3 + 8 * 2
```

dará como resultado 19, no 22, que sería lo que daría si los operadores se aplicaran en el orden en que están escritos, o sea, de izquierda a derecha. Más precisamente, dicho valor 19 se obtiene aplicando en primer lugar el operador *, ya que tiene mayor precedencia, calculándose 8 * 2, lo que nos da el valor 16, y a continuación aplicando el operador +,

empleando el resultado anterior como su operando por la derecha (los operandos, como vemos, pueden ser simples o resultantes de cálculos previos), calculándose 3 + 16, obteniéndose finalmente el valor resultado 19. Este mismo esquema se repetiría para los distintos operadores de acuerdo con las reglas de precedencia en expresiones con más operandos.

No obstante, el orden de las operaciones predefinido se puede modificar usando paréntesis (podemos usar cuantos sean convenientes), forzando en el siguiente ejemplo a calcular primero 3 + 8 a pesar de la menor precedencia de + frente a *:

```
(3 + 8) * 2
```

El siguiente listado muestra parte del orden de precedencia predefinido para los operadores de Python, donde x indica el operando para los operadores *unarios*, y los demás son *binarios*, esto es, requieren de dos operandos (los operadores binarios típicamente son *infijos*, o sea, van entre sus operandos, p.e. 0 not in v).

Cuanto **más arriba mayor precedencia**:

```
**
+x, -x, ~x
*, /, //, %
+, -
<<, >>
&
^
|
==, !=, >, >=, <, <=, is, is not, in, not in
not x
and
or
```

Obsérvese que algunos operadores comparten símbolo, p.e. el -, que puede ser unario (alta precedencia) o binario (de algo menor precedencia). Por tanto, suponiendo que la variable n tenga el valor 2, la expresión -n-3 se calcula aplicando primero el - de la izquierda (que es unario pues no tiene operando a su izquierda, y por tanto tiene mayor precedencia), obteniendo el valor -2, y a continuación aplicando el - binario restándole 3, resultando -5. Naturalmente, podríamos escribir -(n-3), con lo que forzamos primero el cálculo entre paréntesis, obteniendo el valor -1, a lo que le aplicamos el - unario resultando 1. Las reglas de precedencia aplican siempre, por tanto incluso podríamos escribir 3*-n, y como el - es unario (a su izquierda no tiene operando, sino el operador *), tiene mayor precedencia que *, por lo que se obtendría primero el valor -2, realizándose a continuación la multiplicación, resultando -6.

## Asociatividad de operadores

La siguiente lista muestra la **precedencia** relativa de parte de los operadores en Python (x indica el operando para los operadores unarios):

```
**
+x, -x, ~x
*, /, //, %
+, -
<<, >>
&
^
|
==, !=, >, >=, <, <=, is, is not, in, not in
not x
and
or
```

Como se puede observar, hay algunos operadores que tienen igual precedencia y que naturalmente podrían aparecer en una misma expresión. También puede ocurrir que el mismo operador (p.e. el - binario) aparezca varias veces en la expresión, lógicamente en todas las ocurrencias con su (igual) precedencia. En estos casos de igual precedencia esta no nos sirve para determinar completamente el orden de cálculo, por lo que se acude a las reglas de **asociatividad**. La gran mayoría de los operadores binarios son **asociativos por la izquierda**, o sea, dentro de un mismo nivel de precedencia se aplican empezando por el que esté más a la izquierda, y siguiendo hacia la derecha. Ejemplo:

> 9 - 2 * 2 + 3  es igual (* mayor precedencia) a 9 - 4 + 3, que es igual (- binario asociativo por la izquierda) a 5 + 3, que es igual a 8.

Si la asociatividad de los operadores binarios + y - fuese de derecha a izquierda, el resultado de la expresión anterior sería:

> 9 - 2 * 2 + 3 es igual a 9 - 4 + 3, que sería igual a 9 - 7, que sería igual a 2: ¡un disparate!

Por conveniencia, el operador de exponenciación en particular es asociativo por la **derecha**:

> 2 ** 3 ** 2 es igual a 2 ** 9 que es igual a 512 (interpretamos que el 2 de la izquierda está elevado a 3-elevado-a-2)

Si ** tuviera asociatividad de izquierda a derecha el resultado sería:

> 2 ** 3 ** 2 sería igual a 8 ** 2 que sería igual a 64: no habríamos considerado que 3 está elevado a 2)

Tiene poco sentido que los operadores de comparación sean asociativos, y en Python se interpretan de forma especial, p.e.:

> a < b < c == d **se interpreta** como sigue (si bien b y c, aunque se muestren repetidos, se evaluarían una sola vez):
>
> a < b and b < c and c == d

Obsérvese que los and tienen menor precedencia y sí son asociativos (por supuesto por la izquierda, además de cortocircuito).

# Ejecución alternativa

## Sentencias de control

Prácticamente todos los programas deben contemplar distintas situaciones o variabilidad en los datos, que requerirán realizar acciones distintas según el caso. Pero esto no sería posible si los programas únicamente pudieran ejecutar sus instrucciones una tras otra, de la primera a la última.

Por tanto, los lenguajes de programación, además de las instrucciones básicas, disponen de estructuras y **sentencias de control** que permiten reorientar *el flujo de ejecución* hacia unas u otras instrucciones para así adaptarse a las distintas situaciones previstas.

En particular, existen sentencias de control que permiten elegir, en función de una (o más) condiciones, entre dos (o más) posibles "vías" por los que continuar la ejecución; y otras que permiten ejecutar repetidamente un mismo grupo de instrucciones, ya sea un cierto número de veces o mientras se cumpla una condición. En esta lección nos centraremos en las primeras.

Una **condición** es cualquier expresión que al ser evaluada dé un resultado booleano: en Python un valor **True** o **False** que nos indicará, respectivamente, si se ha cumplido o no la condición.

## Condiciones

Las sentencias de control suelen decidir las acciones a ejecutar en función de condiciones, que son expresiones que al ser evaluadas devuelven un resultado booleano.

Para formar condiciones se usan con frecuencia los **operadores relacionales**, que son operadores que comparan los valores de sus dos operandos y devuelven un valor booleano de acuerdo con la relación entre ellos.

Supongamos, en los siguientes ejemplos, que la variable *a* tiene asociado el valor 10 y la variable *b* tiene asociado el valor 20:

| operador | descripción | ejemplo |
|---|---|---|
| **==** | Compara si los operandos **son iguales** entre sí. | a == b *da falso* |
| **!=** | Compara si los operandos **no son iguales** entre sí. | a != b *da verdadero* |
| **>** | Compara si el operando izquierdo es **mayor** que el derecho. | a > b *da falso* |
| **<** | Compara si el operando izquierdo es **menor** que el derecho. | a < b *da verdadero* |
| **>=** | Compara si el operando izquierdo es **mayor o igual** que el derecho. | a >= b *da falso* |
| **<=** | Compara si el operando izquierdo es **menor o igual** que el derecho. | a <= b *da verdadero* |

(En los símbolos de operador formados por dos caracteres, estos deben escribirse siempre juntos y en el orden indicado).

Se pueden formar condiciones más complejas combinando el resultado de condiciones simples usando **operadores booleanos**. Suponiendo que x e y sean valores booleanos:

| operación | resultado |
|---|---|
| x **or** y | da False si ambos son False, si no, da True |
| x **and** y | da True si ambos son True, si no, da False |
| **not** x | da False si x es True, si no, da True |

Ejemplos de condiciones compuestas:

```
x > y and y < z
```

```
(x > y or x > z) and y > z
```

```
x > 2 and 0 == x % 2
```

Téngase en cuenta que, aparte paréntesis, las operaciones aritméticas tienen precedencia sobre las comparaciones y estas sobre las operaciones booleanas. Por ejemplo, supongamos que x vale 3 al momento de evaluar la última expresión: primero se evalúa el operando izquierdo del and, para ello se compara si x es mayor que 2, resultando True, por lo que a continuación se evalúa el operando derecho, para lo cual primero se calcula el módulo (resto de dividir x entre 2), que da 1 como resultado, valor que se compara en igualdad con el 0, lo que da False; por lo que el resultado del and y, por tanto, de la expresión completa es el valor False.

## Sentencia if

La sentencia **if** permite decidir entre ejecutar, o no, una acción (o una secuencia de acciones) dependiendo de una condición.

```
num = int(input("Dame un número entero: "))
```

```
if num < 0:
    num = -num

print("Elvalor absoluto del número es ", num)
```

En Python la sentencia if empieza con la palabra if seguida de una condición y del caracter de dos puntos, controlando la ejecución de una o más instrucciones en las siguientes líneas; en el ejemplo, únicamente de la instrucción num = -num, que se ejecutará, o no, si la condición num < 0 resulta verdadera o falsa, respectivamnnte.

En general, la sentencia de control **if** se usa para decidir en cada ocasión si se ejecuta, o no, un bloque de instrucciones. Si la decisión fuese no ejecutar dichas instrucciones, estas se "saltan", reanudando la ejecución del programa en la instrucción siguiente a ellas; si por el

contrario se ejecutaran, posteriormente se continuaría la ejecución igualmente en la instrucción siguiente a ellas.

En todas las sentencias de control de Python, las instrucciones que controla se empiezan a escribir a partir de la siguiente línea y aumentando en uno el nivel de **sangrado** respecto a la sentencia de control (con un salto de tabulación o, preferiblemente, **cuatro espacios**).

Nótese, que la instrucción *print* del final del ejemplo, no forma parte de las instrucciones controladas por el if, ya que está al mismo nivel de sangrado que éste, por lo que forma parte de la secuencia primaria del programa.

## Cláusula else

La sentencia if se usa para decidir si se ejecuta, o no, un bloque de instrucciones. Si se decide no ejecutarlas, se saltan y la ejecución continúa en la instrucción siguiente a ellas; si se ejecutan, se alcanza ese mismo punto y se continúa.

Se puede añadir una cláusula **else** a una sentencia if. Una sentencia **if-else** se usa para decidir cuál bloque de instrucciones ejecutar de entre dos posibles alternativas, reanudándose la ejecución en cualquiera de los dos casos a partir de la instrucción que sigue al segundo bloque.

```
num = int(input("Dame un número entero: "))

if num < 0:
   print(num, "es un número negativo")
else:
   print(num, "es un número no negativo")

print("Fin de la ejecución.")
```

En el ejemplo anterior se muestra un mensaje diferente dependiendo de si el valor numérico en la variable num recibido de teclado es o no negativo.

## "Anidamiento"

Las sentencias de control se pueden "anidar", esto significa que una (o más) de las instrucciones controladas por una sentencia de control puede ser a su vez una sentencia de control que controle una subsecuencia de instrucciones, y así sucesivamente.

```
num = int(input("Dame un número entero: "))

if num < 0:
   if num % 2 == 0:
      print(num, "es un número negativo par")
   else:
      print(num, "es un número negativo impar")
```

```
else:
    if num == 0:
        print(num, "es cero")
    else:
        if num % 2 == 0:
            print(num, "es un número positivo par")
        else:
            print(num, "es un número positivo impar")
```

```
print("Fin de la ejecución")
```

En el ejemplo, se empieza comprobando si num es menor que cero; si lo es, se evalúa si es divisible por 2 (el resto de la división es cero), mostrando un mensaje adecuado en función de la respuesta.

Si, por el contrario, el número resultase no ser menor que cero, se comprobaría a continuación si es igual a cero.

Y si al final el número resultara ser mayor que cero, se vuelve a comprobar si es par o impar. Nótese que solo se alcanzará el último de los if (que, por cierto, está en un segundo nivel de anidamiento y por tanto de sangrado) si se ha comprobado previamente que el número no es negativo ni cero, por lo que en ese punto sabemos que el número no puede sino ser positivo. Y en el último de los else sabemos además que el número no es par, por lo que se puede realizar la acción correspondiente sin más comprobaciones.

## Contracción elif

Cuando se encadenan una o más cláusulas else conteniendo solo una sentencia if, se pueden contraer en cláusulas **elif** facilitando la sintaxis, seleccionando entre múltiples alternativas excluyentes entre sí, que colocaremos en el mismo nivel de sangrado que el if inicial:

```
num = int(input("Dame un número entero: "))
```

```
if num < 0:
    if num % 2 == 0:
        print(num, "es un número negativo par")
    else:
        print(num, "es un número negativo impar")
elif num == 0:
    print(num, "es cero")
elif num % 2 == 0:
    print(num, "es un número positivo par")
else:
    print(num, "es un número positivo impar")
```

```
print("Fin de la ejecución")
```

Nótese que un if anidado dentro de la cláusula if no se puede abreviar de esta manera; ni tampoco si en una cláusula else a un primer if anidado siguen otras instrucciones en el mismo bloque.

Obsérvese que de la estructura if-elif··elif-else solo se podrá ejecutar uno de sus bloques de instrucciones (o incluso posiblemente ninguno si no hay un else al final). En particular, se ejecutará únicamente el primer bloque cuya condición se cumpla, o el del else si este existe y ninguna condición se cumple, reanudándose (sin comprobar las siguientes condiciones) la ejecución a continuación del último bloque.

A la hora de escribir las condiciones en los elif, hemos de ser conscientes de que para que se pueda alcanzar a comprobar una de ellas, todas las de if y elif anteriores de esa misma estructura han de haber resultado falsas, por lo que sería innecesario volver a comprobarlo. En el ejemplo, al llegar al segundo elif ya se ha comprobado anteriormente que el número no es ni negativo ni cero, de donde podemos concluir que alcanzado ese punto solo puede ser positivo, y por tanto no hace falta comprobarlo.

Una función es un trozo de código organizado para realizar una tarea determinada y con un nombre que permite **llamarla** (invocar su ejecución). Una función se ejecuta solo cuando se la llama, y puede llamársela cuantas veces convenga, y desde múltiples puntos de un programa.

```
def greetings_func():
    """Muestra un saludo."""
    print("Hola")
```

El ejemplo anterior muestra la **definición** de una función en Python, que consta de una **cabecera** y de un **cuerpo**. La cabecera empieza con la palabra clave ***def*** seguida del **nombre** de la función. En el ejemplo, el nombre de la función es *greetings_func*. Los paréntesis que van después son necesarios. La cabecera de la función acaba con el carácter dos puntos.

El cuerpo de la función comienza en la línea siguiente a la cabecera y está desplazado un nivel de sangrado (un tabulador, o, preferiblemente, cuatro espacios) respecto a la misma. Estará formado por cualquier combinación de instrucciones, incluyendo sentencias de control, para realizar la tarea que deseemos.

Como primera línea del cuerpo de una función, justo debajo de la cabecera, es habitual poner un breve comentario en forma de una string encerrada entre **comillas triples**. Este comentario es una ***docstring***, una string de documentación, que se usa para proporcionar información básica sobre la función. Esta información puede ser usada por las aplicaciones de programación para proporcionar ayuda automática y autocompletado de código al usar la función. Se puede acceder a la *docstring* de una función usando el ***atributo* __doc__** tal como se muestra en el siguiente ejemplo:

```
print(greetings_func.__doc__)
```

El resultado del código anterior es:

```
Muestra un saludo
```

En general, los literales de string con comillas triples ocupan varias líneas:

```
    """Esto es un ejemplo de docstring multilínea.
       Las comillas de apertura y cierre
       deben estar alineadas.
    """
```

Para usar una función hay que invocarla (llamarla). La **llamada** a una función es como una instrucción más, que se forma poniendo el nombre de la función seguido de paréntesis:

```
def greetings_func():
    """Muestra un saludo"""
    print("Hola")
```

```
print("Antes dellamar a greetings_func")
greetings_func()  # Llamada a greetings_func
print("Después dellamar a greetings_func")
```

Al terminar la ejecución de una función, el flujo de ejecución **retorna** al punto justo después de la llamada, reanudándose la ejecución a partir de ahí. En cierto sentido, definir una función es como si hubiésemos añadido una nueva instrucción u operación al lenguaje, y la llamada sería su ejecución.

El resultado del código anterior es, pues:

```
Antes dellamar a greetings_func
Hola
Después dellamar a greetings_func
```

# Funciones

## Funciones con parámetros

Las funciones pueden tener **parámetros**, que en Python funcionan como variables internas de la función durante su ejecución. Los parámetros de la función se inicializan con los datos pasados en la llamada. (datos de entrada necesarios para realizar la tarea). Por ejemplo:

```
def greetings_func(name):
    """Muestra un saludo.
```

```
    Parameter name: str - nombre de la persona a saludar
    """
    print("Hola", name)
```

La función greetings_func tiene ahora un parámetro, en este caso de nombre name, que se usa para incluir en el saludo el nombre de la persona a la que se saluda. Una función puede tener varios parámetros, cuyos nombres se escriben dentro de los paréntesis que siguen al nombre de la función, formando la denominada lista de **parámetros formales**. La lista de parámetros formales indica qué datos espera recibir la función cuando se la llame para realizar su tarea.

```
def consumo_medio(kilometros, litros):
    ...
```

Al llamar a una función que tiene parámetros, hay que pasarle **valores** para esos parámetros; estos valores se denominan **parámetros reales** (*actual parameters* en inglés, a veces llamados también *arguments* "argumentos"). El nombre de parámetros reales viene del hecho de que son los valores reales que se usan en la función en cada ocasión, frente a los formales, que son indicadores de los parámetros que se necesitan, pendientes de concretar en cada llamada:

```
name1 = "Pedro"
name2 = "Juan"
greetings_func(name1)
greetings_func(name2)
greetings_func("María" + " " + "Elena")
```

Como muestra el ejemplo, los parámetros reales pueden ser variables, valores literales, o, en general, cualquier expresión que dé como resultado un valor del tipo adecuado.

En el siguiente ejemplo, la función input devuelve una string (que ha tecleado el usuario), la cual a su vez la recibe greetings_func en su parámetro name:

```
greetings_func(input("Hola, ¿cómo te llamas? "))
```

Si la función tiene varios parámetros, hay que pasar valores para todos ellos; en Python, esto se puede hacer **por posición** (un parámetro real se asocia con el parámetro formal que ocupa su misma posición en la lista de parámetros) o **por nombre** (un parámetro real se asocia explícitamente con el nombre de un parámetro formal). El paso **por posición** es el habitual en la mayoría de los lenguajes de programación:

```
def consumo_medio(kilometros, litros):
    ...
```

```
consumo_medio(100, 6)  # Paso por posición → (kilometros = 107, litros = 6)
consumo_medio(litros=17.5, kilometros=197)  # Paso por nombre
```

(Obsérvese que, tanto para los parámetros formales como para los actuales, se suele dejar un espacio después de la coma).

Los parámetros formales pueden tener **valores por omisión**, que se usan si en la llamada se omite dar un valor para ese parámetro. Los parámetros que no tienen valores por omisión no pueden omitirse en la llamada.

```
def consumo_medio(distancia, litros, unidades="kilometros"):
    ...
```

```
consumo_medio(103, 6.5)  # distancia = 103, litros= 6.5, unidades = "kilometros" (valor por omisión)
consumo_medio(87, 11, "millas")  # distancia = 87, litros= 11, unidades = "millas"
```

Los parámetros reales que estén después de uno o más que "se pasan" por omisión tienen que pasarse obligatoriamente por nombre, ya que se pierde la correspondencia entre la lista de parámetros formales y la lista de parámetros reales.

Es importante comprender que la relación en Python entre parámetros reales y formales consiste en que de los reales se toman sus valores, los cuales se asignan al comienzo de la ejecución de la llamada a los parámetros formales, que son variables internas de la función, que desaparecen al retornar la ejecución de esta. El que en una llamada usemos variables para algunos parámetros reales es irrelevante (lo que nos importa es el valor que contienen en el momento de la llamada) en el sentido de que estas variables no tienen relación con los parámetros formales, incluso si les diésemos el mismo nombre, p.e. en una llamada f(x), x será una variable en el contexto en que se genera la llamada, y aunque al parámetro formal de f podríamos haberlo llamado también x "por coincidencia", no existe confusión ni conflicto, pues este parámetro formal x es una variable interna a la función distinta de la otra variable también llamada x existente en el punto de la llamada, de forma que incluso si durante la ejecución de la función le asignáramos un nuevo valor al parámetro formal x, la otra variable x no se vería afectada.

## Funciones que devuelven un resultado

Además de poder admitir datos como parámetros, las funciones pueden devolver un resultado al finalizar su tarea. Esto es coherente (además del nombre) con que las funciones resuelven problemas del tipo hallar un dato desconocido a partir de otros datos conocidos, aplicando un **algoritmo**, que describe la solución de un problema en función de los datos necesarios para representar un caso concreto del problema y de los pasos necesarios para obtener el resultado deseado.

Para devolver un resultado se utiliza la instrucción ***return*** con el valor a devolver (el cual se puede expresar en forma de un literal del tipo apropiado, una variable que lo contenga o, en general, cualquier expresión, cuya evaluación nos dará el valor a devolver):

```
def suma(a, b):
    """Suma losvalores de susparámetros."""
```

```
    result = a + b
return result
```

El resultado devuelto por una función se puede usar directamente en una expresión (es como si lo hubiésemos escrito en lugar de la llamada, aunque con la flexibilidad de que dicho valor será uno u otro, p.e. según los valores de los parámetros en ese momento):

```
print(suma(a, b))
...
if 10 * suma(a, b) > 50:
    ...
```

También se puede asignar a una variable para usarlo más tarde:

```
x = suma(a, b)
```

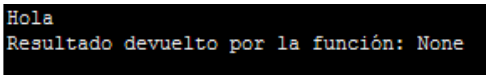
(Recuérdese que, aunque hayamos empleado iguales nombres para las variables usadas como parámetros reales y para los parámetros formales de *suma*, ello no supone que haya relación alguna entre ellas pues, aunque compartan nombre, serán variables distintas).

En realidad, en Python todas las funciones devuelven un valor, solo que las que no lo hacen explícitamente mediante *return* devuelven el valor ***None***, que es el único valor del tipo de datos *NoneType* y se interpreta como "Ningún valor". Por ejemplo, el siguiente código:

```
def greetings_func():
    """Muestra un saludo."""
    print("Hola")

result = greetings_func();
print("Resultado devuelto por lafunción:", result);
```

produce el siguiente resultado:




La palabra "Hola" la escribe la función, que devuelve *None*. El valor *None* devuelto por la función es asignado a la variable *result* y mostrado por la instrucción *print* que se ejecuta en la siguiente línea.

La instrucción *return* puede utilizarse como cualquier otra instrucción dentro de una función, esto es, puede haber varios return en distintos puntos de la misma, eso sí, una vez que se alcanza un return, termina la ejecución de la función (por lo que no tiene sentido escribir una instrucción del mismo nivel de sangrado inmediatamente después).

## Funciones polimórficas

Muchos lenguajes de programación exigen que se declaren los tipos de los datos que pueden asociarse a una variable o pasarse a un parámetro formal o que devuelva una función. Python tiene **tipado dinámico**, lo que significa que no requiere declaraciones de esta clase: el tipo de dato asociado a una variable lo determina el del valor que se e asigna en el momento de crearla, y un parámetro formal puede asociarse a un tipo de dato diferente en cada llamada. Esto, en el caso de las funciones, permite que
sean **polimórficas** por definición (una función polimórfica es aquella que se puede usar con distintos tipos de parámetros y/o distinto número de parámetros). Una función en Python tal como:

```
def sum(a, b):
    """Suma losvalores de susparámetros"""
    result = a + b
    return result
```

puede aplicarse sin problema a cualquier pareja de valores que admitan el uso del operador +, entre ellos:

```
print(sum(1, 3))             # Númerosenteros
print(sum(3.5, 4.8))         # Números reales
print(sum(3.5, 4))           # Un número entero con un real
print(sum("Hola ", "mundo"))  #Strings, que serán concatenadas con el +
print(sum(3+2j, 5-1j))       # Números complejos
```

Resultado de la ejecución anterior:

```
4
8.3
7.5
Hola mundo
(8+1j)
```

## Anotación de los tipos de los parámetros de una función

Aunque el polimorfismo proporcionado a Python por el tipado dinámico puede ser una potente herramienta (permite que una función se use en situaciones muy diferentes, sin tener que reescribir múltiples versiones), cuando no es nuestra intención escribir una función polimórfica puede ser útil explicitar qué tipo de dato se supone que debe recibir un parámetro formal, o qué tipo de resultado se supone que va a devolver una función:

```
def sum(a: int, b: int) → int:
    """Suma losvalores de susparámetros."""
    result = a + b
    return result
```

Como se muestra en el ejemplo anterior, para los tipos más básicos (int, float, bool, str) basta con poner dos puntos detrás del parámetro y añadir el tipo esperado. Para otros tipos de datos, que se irán viendo en otras lecciones, puede ser necesario incluir alguna instrucción adicional. El tipo del resultado de la función se explicita después de una "flecha" añadida al final de la cabecera, antes de los dos puntos.

El primer efecto del tipado explícito es informativo. Normalmente, en la *docstring* de una función se pone información sobre sus parámetros, incluyendo de qué tipo se espera que sean, como en el siguiente ejemplo:

```
"""Suma losvalores de susparámetros.
Parámetros:
  a: (int) - Primer sumando
  b: (int) - Segundo sumando
"""
```

Si se explicitan los tipos en la lista de parámetros formales y, además, se usan nombres adecuados, la *docstring* puede reducirse sustancialmente, ya que mucha información relevante está incluida en la propia cabecera de la función.

```
def sum(primer_sumando: int, segundo_sumando: int) → int:
    """Suma losvalores de susparámetros"""
    result = primer_sumando + segundo_sumando
    return result
```

Además de esto, se pueden usar herramientas específicas que, basándose en la información proporcionada por el tipado explícito, son capaces de detectar por anticipado posibles errores que, de otro modo, sólo se manifestarían al ejecutar el programa.

## Dónde escribir una función

En Python una función se puede escribir junto con el código que la usa, en el mismo archivo:

archivo **my_program.py**

```
def my_func(a: int, b: float) → bool:
    return a > b

num1 = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(my_func(num1, num2))
```

Pero un código puede llamar a una función escrita en otro archivo. Para ello, debe incluir previamente una cláusula de importación:

archivo **my_program.py**

```
import functions
```

```
num1 = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(functions.my_func(num1, num2))
```

archivo **functions.py**

```
def my_func(a: int, b: float) → bool:
    return a > b
```

Esta segunda aproximación independiza a la función del código que la usa, lo que permite que sea usada desde piezas de código escritas en diferentes archivos.

# Módulos y paquetes

## Módulos

La programación modular es una técnica de diseño de software que enfatiza la separación de la funcionalidad de un programa en módulos independientes e intercambiables, de manera que cada uno contiene todo lo necesario para ejecutar sólo un aspecto de la funcionalidad deseada. Por ejemplo, en un juego, podríamos tener, entre otros, un módulo para controlar la lógica del juego, otro para interactuar con el usuario y un tercero para mostrar el escenario de juego en la pantalla.

La programación modular permite gestionar la complejidad, dividiendo una funcionalidad compleja en otras más simples, y permite reusar el código, ya que los módulos deben ser independientes e intercambiables, es decir, un módulo desarrollado inicialmente para un programa podría usarse en otro y un programa que usa un módulo podría cambiarlo por otro mejor.

En Python, cada fichero con extensión **.py** alberga un **módulo**, que se llama como el archivo que lo contiene (sin la extensión). Para usar, desde un módulo distinto, las funcionalidades ofrecidas por un módulo, solo hay que poner su nombre en una cláusula de **importación**:

*módulo* **my_program.py**

```
import functions
```

```
num1 = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(functions.my_func(num1, num2))
print(functions.pi)
```

*módulo* **functions.py**

```
pi = 3.14

def my_func(a: int, b: float) → bool:
    return a > b
```

El módulo functions del ejemplo anterior ofrece dos elementos que se pueden utilizar: la variable pi y la función my_func. Ambos, se pueden utilizar, una vez importado el módulo, escribiendo su nombre precedido del nombre del módulo separado por un punto.

```
print(functions.my_func(num1, num2))
print(functions.pi)
```

Un módulo proporciona un ***namespace***, un espacio de nombres en el que podemos declarar elementos sin entrar en conflicto con elementos de otros módulos que puedan tener el mismo nombre, ya que se diferenciarían al prefijarlos con el nombre del módulo (por ejemplo, podríamos usar en el mismo programa dos módulos que declarasen una función llamada *my_func* sin que se confundan).

En el caso de que un módulo ofrezca varios elementos pero sólo necesitemos usar algunos, los elementos que queramos se pueden importar individualmente, como muestra el siguiente ejemplo:

*módulo* **my_program.py**

```
from functions import my_func

num1 = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(my_func(num1, num2))
```

Los elementos importados directamente no pueden ser prefijados con el nombre del módulo a la hora de usarlos. Para evitar conflictos si queremos usar elementos con el mismo nombre de distintos módulos, podemos, o importar los módulos, para poder prefijar los elementos que queremos usar, o usar **alias**:

```
from module1 import suma as suma1
from module2 import suma as suma2
```

También, se pueden usar alias al importar un módulo completo, generalmente, para usar un nombre más corto y manejable:

```
import functions as funcs
```

## El módulo __main__

Un módulo puede incluir secuencias de instrucciones que no forman parte de ninguna función:

*módulo* **functions.py**

```
def my_func(a: int, b: float) → bool:
    return a > b

print("Esto no forma parte de unafunción")
```

Tales instrucciones "libres" se ejecutan cuando se carga el módulo (la primera vez que es objeto de una importación), a diferencia de las funciones, que sólo se ejecutan cuando se las llama:

*módulo* **my_program.py**

```
from functions import my_func

radi = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(my_func(num1, num2))
```

Resultado:

```
Esto no forma parte de unafunción
Entra un número entero: 1
Entra un número entero: 2False
```

Muchas veces no queremos que este código "libre" se ejecute cuando el módulo es importado, sino solo cuando es el módulo "principal" (inicial) del programa, para lo cual comprobamos si el módulo responde al nombre "__main__":

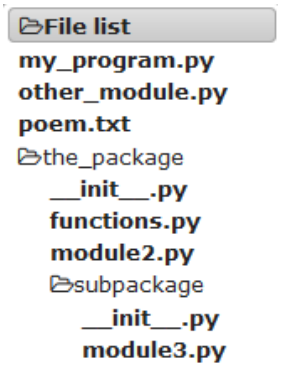
*módulo* **functions.py**

```
def my_func(a: int, b: float) → bool:
    return a > b

if__name__ ==  "__main__":
    print("Esto no forma parte de unafunción")
```

El nombre __main__ (nótense los dos guiones bajos a cada lado) es asignado automáticamente al módulo que inicia la ejecución de un programa. De esta manera, el código controlado por esa sentencia if sólo se ejecuta si el módulo en que está es ejecutado directamente, pero no cuando es importado por otro que se empezó a ejecutar primero.

## Paquetes




En programación modular, el concepto "paquete" (en inglés,

*package*) suele usarse para referirse a una agrupación lógica de módulos: distintas funcionalidades se implementan en diferentes módulos, que pueden incluir funciones, variables, y otros elementos relacionados, y un conjunto de módulos con funcionalidades relacionadas se agrupan en un **paquete**.

En Python, al igual que un módulo se implementa en un archivo, un paquete, que es un conjunto de módulos, y por tanto implica un conjunto de archivos, se corresponde, físicamente, con una carpeta o directorio en el sistema de ficheros.

Para que un directorio sea un paquete debe incluir un "archivo de configuración" llamado **__init__.py**, que puede estar vacío, y que sirve para controlar el acceso a los módulos contenidos en el paquete desde módulos externos.

La figura muestra una estructura formada por dos módulos (*my_program* y *other_module*) que no forman parte de ningún paquete más un paquete de nombre *the_package*, conteniendo este, otros dos módulos (*functions* y *module2*) más un subpaquete de nombre *subpackage,* el cual contiene a su vez un único módulo (*module3*).

Al importar un módulo, hay que especificar el paquete al que pertenece, como se hace en el siguiente ejemplo:

*módulo* **my_program.py**

```
from the_package.functions import my_func

num1 = int(input("Entra un número entero: "))
num2 = int(input("Entra un número entero: "))
print(my_func(num1, num2))
```

# Tuplas

Los tipos básicos de datos (int, float, bool) representan valores monolíticos, que no se pueden separar en componentes individuales con entidad propia. Por su parte las strings (str) representan valores compuestos, formados por una secuencia de caracteres, aunque el

sentido de lo que representan suele venir dado por el conjunto, más que por los caracteres individuales. Los números complejos (complex) también son valores compuestos, formados por una pareja de números reales.

Hay muchas clases de información que se pueden entender como formadas por datos que son un agregado de datos más simples con entidad propia. Ejemplos: un nombre español completo está formado habitualmente por tres strings: nombre de pila, primer apellido y segundo apellido; un punto en el espacio bidimensional se representa por dos coordenadas, que son números reales; una fecha está formada por día, mes y año; etc.

En Python este tipo de información compuesta por un agregado finito de datos se puede representar de manera unitaria usando tuplas (tipo **tuple**). Podemos definir una n-tupla como una secuencia de n valores. En Python podemos construir una tupla encerrando una secuencia de valores, separados por coma, entre paréntesis.

```
name = ("Pedro", "Martel", "Parrilla")
point2D = (3.5, 8.3)
vector3D = (3.5, 8.2, 4.6)
dni = "42387423H"
age = 16
address = "Calle del Pino, nº 2. Las Palmas."
person_id = (dni, name, age, address)
```

El ejemplo anterior construye cuatro tuplas, referenciadas
como: name, point2D, vector3D y person_id. Las variables dni y address referencian strings, no tuplas, y la variable age referencia un valor de tipo int. La variable person_id ilustra que:

1. Los elementos de una tupla pueden ser de tipos diferentes ( está formada solo por strings y  y  solo por números reales, mientras que  está formada por dos strings (dni y address), un número (age) y una tupla (name)

   name

   point2D

   vector3D

   person_id
2. Una tupla puede estar formada por elementos de cualquier tipo , incluso tuplas

Nótese, además, que esta última tupla asignada a person_id **no** contiene las variables dni, name, etc. sino que referencia a los valores que en ese momento estaban referenciando dichas variables. Si posteriormente asignáramos nuevos valores a dichas variables, ello no afectará a la mencionada tupla. De hecho, en general, al construir una tupla podemos indicar para cada elemento una expresión cuyo valor resultante en ese momento será el que corresponda a dicho elemento:

```
a = 2
b = 3
t = (b, 1-b, a+b, a-b)
```

```
b = 7     # modificamos b, pero ello no afecta a la tupla
print(t)  # muestra (3, -2, 5, -1)
```

En fin, en el caso de que queramos formar una tupla con un solo elemento, debemos poner una coma detrás del elemento:

```
name = ("Pedro",)
```

## Acceso a los elementos individuales de una tupla

Una tupla es una secuencia finita de valores:

```
name = ("Pedro", "Martel", "Parrilla")
```

Una secuencia establece un orden en el sentido de que cada elemento ocupa una posición concreta:

```
Posiciones:   1ª      2ª       3ª
name      = ("Pedro", "Martel", "Parrilla")
```

En Python las posiciones de una secuencia se identifican mediante índices secuenciales, que son números enteros. El índice de la primera posición es el 0.

```
Posiciones:   1ª      2ª       3ª
name      = ("Pedro", "Martel", "Parrilla")
Índices   :   0       1        2
```

Para acceder a un elemento individual, basta con poner la tupla seguida del índice de la posición a la que queremos acceder, encerrado entre corchetes.

```
first  = name[0]  # Pedro
second = name[1]  # Martel
third  = name[2]  # Parrilla
```

El índice de la última posición es igual a la longitud de la tupla (número de elementos) menos 1. Recordando esto, también se puede acceder a los elementos de una tupla usando índices negativos relativos a la longitud de la tupla:

```
first  = name[-3]  # Pedro
second = name[-2]  # Martel
third  = name[-1]  # Parrilla (-1 nos sirve siempre como últimoíndice)
```

## Operaciones con tuplas

De una tupla se puede conocer su **longitud** (número de elementos) usando la función predefinida ***len***:

```
name = ("Pedro", "Martel", "Parrilla")
len_name = len(name)  # A len_name se le asigna elvalor 3
```

Se pueden concatenar ("sumar") dos tuplas, obteniéndose una nueva tupla con los valores de una y otra:

```
tupla1 = (1, 2, 3)
tupla2 = (4, 5, 6)
tupla3 = tupla1 + tupla2  # → (1, 2, 3, 4, 5, 6)
```

Se puede "multiplicar" una tupla por un número positivo, lo que equivale a concatenarla tantas veces como indique el número:

```
tupla1 = (1, 2, 3)
tupla2 = tupla1 * 3  # → (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Si se "multiplica" por un número negativo o 0, se obtiene la tupla vacía, que contiene cero elementos y podemos expresar con ().

Se pueden comparar dos tuplas usando los operadores relacionales. La comparación se realiza comparando los elementos de principio a fin; es menor la que primero tenga un elemento menor, o, si todos los elementos son iguales hasta que se acaba una, la que tenga menos elementos.

```
tupla1 = (1, 2, 3)
tupla2 = (1, 3, 4)

iguales           = tupla1 == tupla2  # False
menor_la_1        = tupla1 < tupla2   # True
mayor_o_igual_la_1 = tupla1 >= tupla2  # False
```

## Operaciones con tuplas (2)

Se puede construir una tupla a partir de un segmento de otra:

```
tupla1 = (10, 21, 13, 52, 26, 17, 81, 29)
tupla2 = tupla1[3:6]  # → (52, 26, 17)
```

El segmento deseado se identifica indicando el índice del primer elemento y el índice de la posición siguiente al último elemento. Si se omite el segundo valor, se toma el segmento desde el primer índice hasta el final:

```
tupla1 = (10, 21, 13, 52, 26, 17, 81, 29)
tupla2 = tupla1[3:]  # → (52, 26, 17, 81, 29)
```

Si se omite el primer valor, se toma el segmento desde el principio hasta la posición anterior a la indicada por el índice presente:

```
tupla1 = (10, 21, 13, 52, 26, 17, 81, 29)
tupla2 = tupla1[:6]  # → (10, 21, 13, 52, 26, 17)
```

Si se omiten ambos índices, el segmento coincide con la tupla original completa (se copia la tupla entera):

```
tupla1 = (10, 21, 13, 52, 26, 17, 81, 29)
tupla2 = tupla1[:]  # → (10, 21, 13, 52, 26, 17, 81, 29)
```

Se puede añadir un tercer valor entre corchetes para indicar un incremento o "paso" que permite saltarse elementos. En el siguiente ejemplo se toman los elementos desde el principio hasta el final saltando de dos en dos (índices 0, 2, 4, ⋯ ) con los que crear la nueva tupla:

```
tupla1 = (10, 21, 13, 52, 26, 17, 81, 29)
tupla2 = tupla1[::2]  # → (10, 13, 26, 81)
```

Nótese que todo lo dicho de las operaciones de obtención de la longitud (*len*), concatenación (+), repetición (*) y segmentación ([:] y [::]) es aplicable igualmente a las strings, porque, en realidad, son **operaciones de tratamiento de secuencias, y tanto las strings como las tuplas son secuencias** , en un caso, secuencias de caracteres y, en el otro, secuencias de elementos de cualesquiera tipos.

## Anotación de variables tupla

Para anotar que una variable o un parámetro es de un tipo no básico, hay que importarlo desde el módulo **typing**:

```
from typing import Tuple
```

```
tupla: Tuple         # Lavariable tupla se anota como una tupla (tipo Tuple)
vector: Tuple[float] # Lavariable vector se anota como una tupla de elementos de tipo float
```

A partir de la versión 3.9 de Python, se puede anotar usando el nombre real del tipo (en minúsculas):

```
tupla: tuple         # Lavariable tupla se anota como una tupla (tipo tuple)
vector: tuple[float] # Lavariable vector se anota como una tupla de elementos de tipo float
```

*Función divmod(), es una función incorporada en Python 3, que **devuelve el cociente y el resto al dividir el número a por el número b**. Toma dos números como argumentos a & b . El argumento no puede ser un número complejo.*

# Iteración: while

## Repetición

A veces, un algoritmo requiere repetir una acción o conjunto de acciones varias veces seguidas. Por ejemplo, supongamos que queremos que un programa muestre diez líneas con la palabra "Hola". Es fácil:

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

La repetición, o **iteración**, es bastante frecuente, por lo que los lenguajes ofrecen estructuras para expresarlas de un modo más conciso y flexible:

```
líneas_escritas = 0
líneas_a_escribir = 10
whilelíneas_escritas <líneas_a_escribir:
   print("Hola")
líneas_escritas =líneas_escritas + 1
```

La sentencia ***while*** expresa la repetición de una secuencia de instrucciones mientras se cumpla una condición dada. Más precisamente, primero comprueba la condición a la derecha de la palabra clave *while*: si el resultado fuera False, salta las instrucciones del cuerpo o bloque de instrucciones (que en Python se identifican porque tienen un sangrado extra respecto de la palabra *while*), dando por terminada la ejecución de la sentencia *while*, y por tanto reanudando la ejecución a continuación; pero si el resultado fuera True, entra a ejecutar secuencialmente las instrucciones que estén contenidas en su cuerpo, y una vez que no quedaran más instrucciones por ejecutar, volvería al principio y repetiría el proceso: se evaluaría de nuevo la condición (cuyo resultado puede haber variado entretanto si lo han hecho sus operandos), y si fuera False terminaría y saltaría el bloque de instrucciones y si fuera True entraría a ejecutarlo de nuevo y a continuación de nuevo volvería al principio, etc.

etc., repitiéndose el mismo mecanismo cuantas veces fuera necesario en tanto la condición no resulte False. Obsérvese que el número de iteraciones puede ser cualquiera, de 0 en adelante.

Por lo dicho anteriormente, la condición y las instrucciones relacionadas con una sentencia *while* deben estar diseñadas de tal manera que busquen que la condición deje de cumplirse oportunamente en algún intento de iteración. De lo contrario, se entraría en un **bucle infinito**.

En el ejemplo esto lo conseguimos incrementando dentro del cuerpo (y por tanto en cada iteración) la variable *líneas_escritas*, que actúa como contador de las líneas escritas; de esta forma, en algún momento su valor llegará a igualar el de la variable *líneas_a_escribir*, que define el objetivo del problema; la condición dejará entonces de cumplirse y cesará la repetición. Si, por ejemplo, hubiera código posterior al mismo nivel de la palabra *while*, este pasaría a ejecutarse a continuación.

## Repetición (2)

Una sentencia de repetición, como while, permite expresar la repetición de una secuencia de instrucciones de forma concisa.

```
líneas_escritas = 0
líneas_a_escribir = 10
whilelíneas_escritas <líneas_a_escribir:
    print("Hola")
líneas_escritas =líneas_escritas + 1
```

Sin embargo, la mayor virtud de una sentencia de repetición no es la concisión, sino la flexibilidad que proporciona. El siguiente ejemplo es una variante del anterior en la que el objetivo de líneas a escribir no depende de un valor fijo, sino que puede variar en cada ejecución, produciendo resultados diferentes en cada caso, sin tener que modificar el algoritmo:

```
líneas_escritas = 0
líneas_a_escribir = int(input("¿Líneas a escribir? "))
whilelíneas_escritas <líneas_a_escribir:
    print("Hola")
líneas_escritas =líneas_escritas + 1
```

En el caso de que el usuario introduzca un valor igual o menor que cero, no se escribirá ninguna línea, ya que la condición no se cumple desde el principio; en cualquier otro caso, se mostrarán tantas líneas como haya indicado el usuario.

## Esquema básico de iteración

La iteración expresada por la sentencia while responde a un esquema básico bastante simple:

```
inicializaciones
```

```
whilecondición:
Hacer algo (Tratamiento)
avanzar
```

```
líneas_escritas = 0  # Inicializaciones
líneas_a_escribir = int(input("¿Líneas a escribir? "))

whilelíneas_escritas <líneas_a_escribir:  # Condición
    print("Hola")                         # Tratamiento
líneas_escritas =líneas_escritas + 1   # Avance
```

Las inicializaciones, la condición, lo que se hace en cada iteración y cómo se avanza para preparar la siguiente pueden tener muchas variantes, pero siempre es posible reconocer el esquema.

**Ejemplo 1:**

```
current = 2
last = 15
sum_evens = 0
```

```
while current < last:
    sum_evens += current
    current += 2
```

**Ejemplo 2:**

```
penúltimo = 1
último = 1
actual = último
objetivo = 15
iteración = 1
```

```
whileiteración <= objetivo:
    actual = último + penúltimo
    penúltimo = último
    último = actual
iteración += 1
```

**Ejemplo 3:**

```
num1 = 128
num2 = 96
```

```
while num2 > 0:
    mod = num1 % num2
    num1 = num2
    num2 = mod
```

*ABS **calcula el valor absoluto de un número**, donde ese número puede ser un valor literal o una expresión que tome el valor de un número.*

# Iteración: for

## Sentencia for

Un esquema de iteración consiste en repetir una secuencia de acciones mientras se cumpla una condición. En una variante común del esquema, la condición deja de cumplirse cuando se han realizado un número de iteraciones que puede calcularse a priori con facilidad. El siguiente ejemplo muestra, de menor a mayor, los números enteros positivos menores que uno dado (límite), para lo que el bucle debe iterar límite - 1 veces:

```
actual = 1
límite = int(input("Dame unvalor entero positivo: "))

while actual < límite:
    print(actual)
    actual = actual + 1
```

Dada la frecuencia de esta variante, existe una sentencia de repetición específica, que permite expresarla de forma más concisa que la sentencia while:

```
límite = int(input("Dame unvalor entero positivo: "))for actual in range(1, límite):    print(actu
al)
```

```
líneas_a_escribir = int(input("¿Líneas a escribir? "))

forlíneas_escritas in range(líneas_a_escribir):
    print("Hola")
```

Básicamente, una variable de control (*actual*, *líneas_escritas*) va tomando secuencialmente valores de un rango y, cada vez que toma un nuevo valor, se ejecutan las instrucciones anidadas en la sentencia *for*. Nótese que si se pone un solo valor, como en el ejemplo de líneas_a_escribir, el rango va desde cero hasta el valor previo al especificado.

```
range(10)    → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

```
range(1, 10) → 1, 2, 3, 4, 5, 6, 7, 8, 9
```

La función *range* define, en principio, una secuencia de números enteros consecutivos, pero admite un tercer parámetro que permite establecer un "paso" o distancia entre los valores sucesivos. El siguiente ejemplo muestra los números impares positivos menores que 15, por lo que la variable de control va saltando de 2 en 2 a partir del valor inicial:

```
for actual in range(1, 15, 2):
    print(actual, end = ', ')
```

Se muestra: 1, 3, 5, 7, 9, 11, 13,

El paso puede ser negativo, siempre que el rango esté invertido. En el siguiente ejemplo, se muestran, de mayor a menor, los números impares positivos menores o igual que uno dado (el valor inicial es top, o top - 1, si top es par, y la variable de control avanza de -2 en -2):

```
top = int(input("Dame un número entero positivo: "))
```

```
if top % 2 == 0: # Nos aseguramos de empezar convalor impar
    top -= 1
```

```
for actual in range(top, 0, -2):
    print(actual)
```

Si se introduce en la entrada el 8, se muestra:

7

5

3

1

Obsérvese que *range* puede suministrar cero valores —
ejemplos: range(0), range(-4), range(2, 2), range(3, 2), range(2, 5, -1)—, por lo que el *for* no realizaría iteración alguna, terminando inmediatamente.

## Equivalencia entre for y while

La sentencia *for* es más específica que la *while*; es decir, todo lo que se puede hacer usando la sentencia *for* se puede hacer, aunque de forma menos concisa, usando la sentencia *while*. El siguiente ejemplo muestra la equivalencia entre la sentencia *for* y la sentencia *while*:

| Sentencia for | Sentencia while |
| --- | --- |
| `for value in range(first, last, step):` *hacer algo* | `value = first`<br>`while value < last:` *hacer algo* `value += step` |

La principal diferencia entre ambas versiones es que en el *for* la inicialización de la variable de control y su incremento al final de la iteración son automáticos, mientras que en el *while* hay que indicarlos de forma explícita.

Todo lo que se puede hacer con un *for* se puede hacer con un *while*, pero la recíproca no es cierta. Por ejemplo, el bucle *while* del siguiente ejemplo no puede sustituirse por un *for* ya que no se conoce por adelantado el rango de valores que va a tomar la variable num2 hasta llegar a cero (calcularlo costaría tanto como ejecutar el algoritmo):

```
num1 = int(input("Dame un número entero positivo: "))
num2 = int(input("Dame un número entero positivo menor que el anterior: "))
```

```
while num2 > 0:
    mod = num1 % num2
    num1 = num2
    num2 = mod
```

## La sentencia for y el tratamiento de secuencias

La función *range* del siguiente ejemplo define una secuencia de números enteros que debe ir tomando la variable de control del bucle *for*:

```
for actual in range(1, 15):
    print(actual)
```

En realidad, la sentencia *for* admite cualquier clase de secuencia. En el siguiente ejemplo se muestran los días de la semana que han sido almacenados previamente en una tupla:

```
days = ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo")

for day in days:
    print(day)
```

El siguiente ejemplo cuenta el número de letras 'e' que hay en la string referenciada por la variable text:

```
text = "Dame un número entero positivo: "
count = 0

for char in text:
    if char == 'e':
        count += 1

print(count)
```

El bucle *for* requiere que una variable de control recorra una secuencia de valores; da igual el tipo de secuencia de que se trate.

En general, las secuencias pueden contener cero elementos, en cuyo caso esta forma de *for* no realizaría iteración alguna, terminando inmediatamente.

---

Hay que aclarar que el operador *in*, cuando se usa como en los ejemplos anteriores indica el acceso a cada uno de los elementos de una secuencia, pero también puede usar en una condición para evaluar la pertenencia de un elemento a una secuencia/conjunto/contenedor, como en le siguiente ejemplo, que evalúa si la letra 'e' está en un texto:

```
if 'e' in text:
    print(True)
else:
    print(False)
```

## Acceso a los índices al iterar sobre una secuencia

Nótese, que al iterar sobre una secuencia se va accediendo a sus elementos pero no se tiene información sobre la posición que ocupan dichos elementos en la secuencia.

```
days = ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo")

for day in days:
    print(day)
```

Se muestra:

lunes

martes

miércoles

jueves

viernes

sábado

domingo

Si fuese necesario conocer el índice de los elementos, se puede usar la función *enumerate*, e iterar usando una pareja de variables:

```
days = ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo")

for num_day, day in enumerate(days):
    print(num_day, "-", day)
```

Se muestra:

0 - lunes

1 - martes

2 - miércoles

3 - jueves

4 - viernes

5 - sábado

6 - domingo

Básicamente, lo que hace *enumerate* es tomar un objeto iterable (que se puede "recorrer") y añadirle un contador, devolviendo una nueva secuencia de elementos formados por parejas índice, valor.

Se puede obtener el mismo resultado iterando sobre el rango de índices de la secuencia a recorrer, como en el siguiente ejemplo:

```
days = ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo")

for num_day in range(len(days)):
    print(num_day, "-", days[num_day])
```

# Esquemas comunes de tratamiento de secuencias

## Esquemas de recorrido y acumulación

Muchísimos problemas requieren el tratamiento de secuencias de valores aplicando una acción de forma repetida a cada uno de los valores de la secuencia, lo que se hace siguiendo los esquemas básicos determinados por un bucle *while* o un bucle *for*:

| Bucle for | Bucle while |
| --- | --- |
| `for item in range number s: Hacer algo` | `Inicializaciones`<br><br>`while condición: Hacer algo Avanzar` |

Sobre este esquema básico se pueden hacer múltiples variaciones. A modo de ilustración, el Ejemplo 1 muestra un **esquema de recorrido** y el Ejemplo 2 un **esquema acumulador**.

## Ejemplo 1: esquema de recorrido

| Bucle for | Bucle while |
| --- | --- |
| number s = (21, 13, 24, 42, 12)<br>for<br>item in number s: print( item ) | number s = (21, 13, 24, 42, 12)index = 0<br><br>while index < len( number s): print( number s[index]) index += 1 |

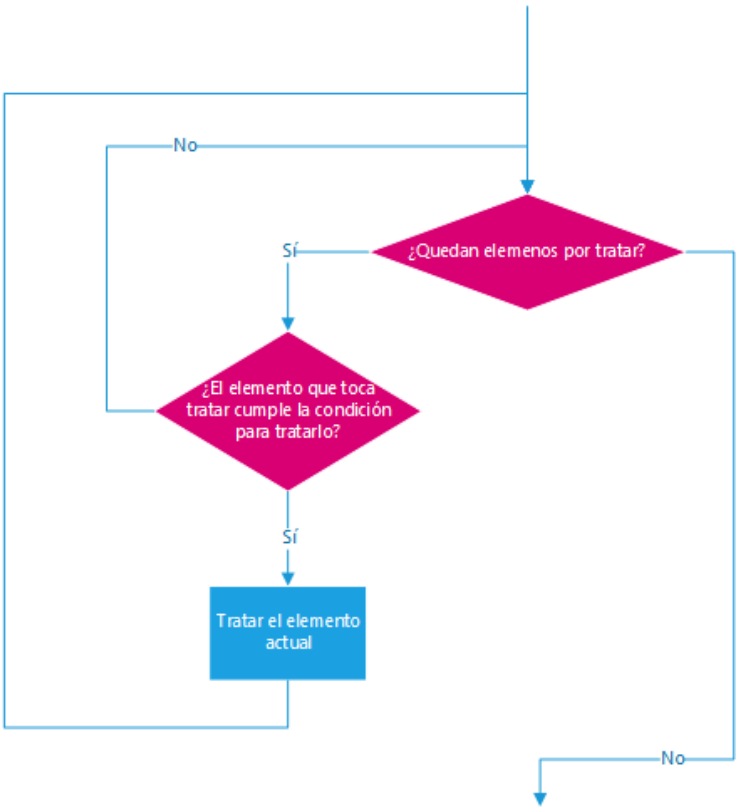
## Ejemplo 2: esquema de acumulación

| Bucle for | Bucle while |
| --- | --- |
| number s = (21, 13, 24, 42, 12)<br><br>addition = 0<br>for<br>item in number s: addition +=<br>item print(addition) | number s = (21, 13, 24, 42, 12)index = 0addition = 0<br><br>while index < len( number s): addition += number s[index] index += 1print(addition) |

La diferencia entre el primer y segundo esquema es que el de recorrido trata una secuencia realizando una acción con cada uno de sus elementos, acción que sólo depende del elemento en cuestión y no de ningún otro elemento de la secuencia, mientras que en el esquema de acumulación la acción va combinando los elementos para obtener un valor final, una vez acabado el bucle. Habitualmente se asocia el esquema de acumulación con cualquier operación que tenga elemento neutro (para inicializar el proceso) y sea asociativa, como la adición o la multiplicación numéricas, o la concatenación de strings; pero podríamos generalizarlo para cubrir cualquier algoritmo en que se calcule un dato en función del conjunto de elementos de una secuencia, combinando los valores de los elementos de alguna forma.

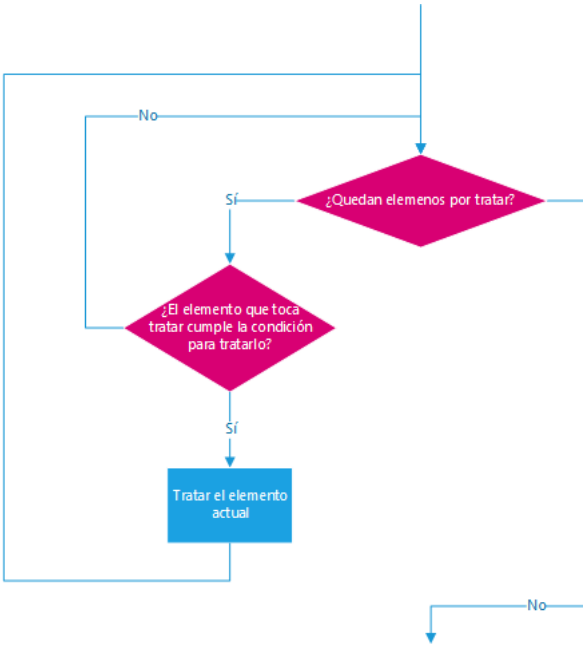
Combinando la estructura de repetición con otras sentencias de control se pueden obtener esquemas más sofisticados.

## Esquema de selección




Combinando un bucle con una sentencia de selección se obtiene un **esquema de selección** como el del siguiente ejemplo, que muestra sólo los valor es impares de una tupla de números enteros.

| Bucle for | Bucle while |
| --- | --- |
| `number s = (21, 13, 24, 42, 12) for number in number s: if number % 2 == 1: print( number )` | `number s = (21, 13, 24, 42, 12)index = 0`<br><br>`while index < len( number s): number = number s[index] if number % 2 == 1: print( number ) index += 1` |




Un esquema de selección, como su nombre indica, aplica una acción sólo a los elementos de la secuencia que cumplen una condición determinada. También podría considerarse la variante de aplicar un tratamiento a los valores que cumplen una condición y otro distinto, a los que no la cumplen.

| Bucle for | Bucle while |
| --- | --- |
| number s = (21, 13, 24, 42, 12)<br>for number in number s: if number % 2 == 1: print( number )<br>else:<br>print("*") | number s = (21, 13, 24, 42, 12)index = 0<br><br>while index < len( number s): number = number s[index] if number % 2 == 1: print( number ) else: print("*") index += 1 |

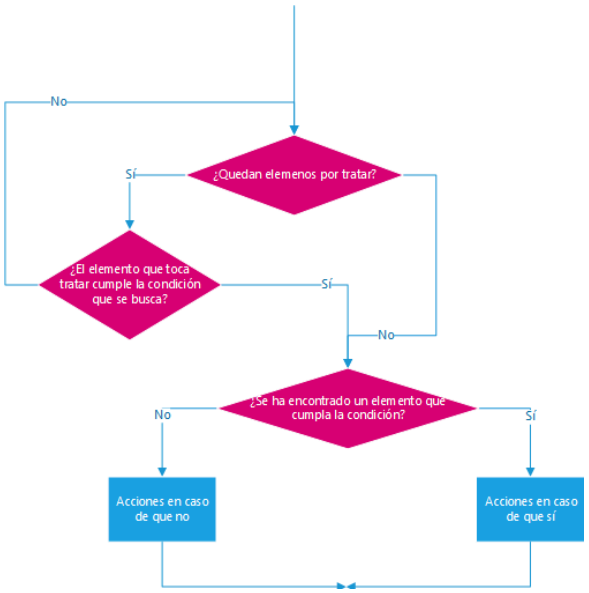
## Esquema de búsqueda

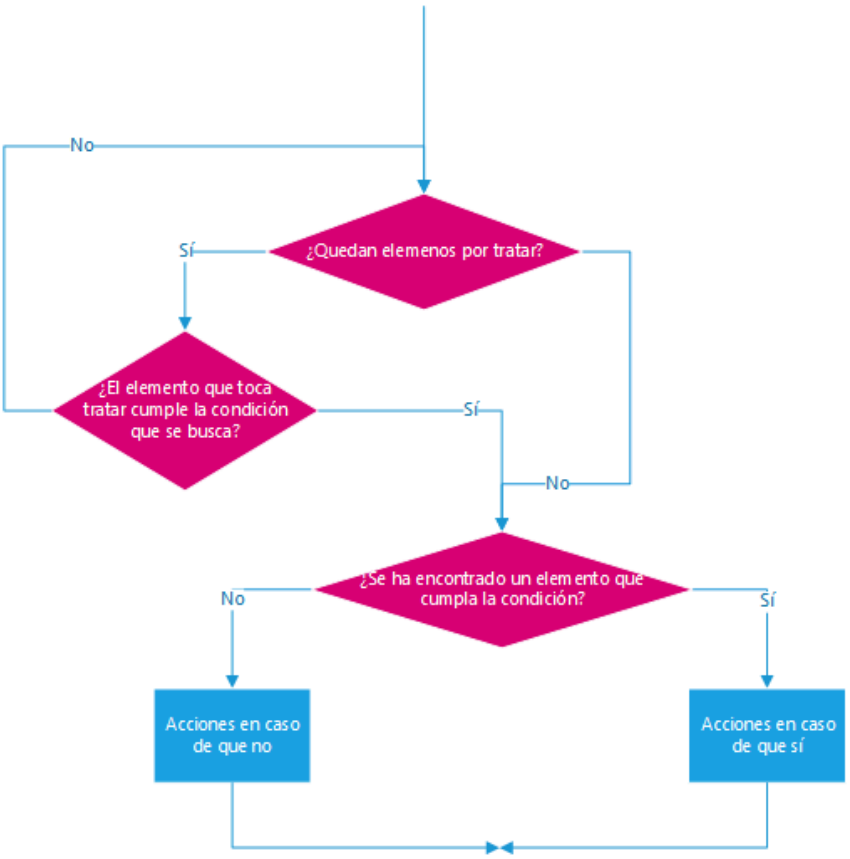
El siguiente ejemplo localiza el primer número par en una tupla de números enteros:

| Bucle for | Bucle while |
| --- | --- |
| number s = (19, 12, 35, 21, 12, 41, 37)<br>found = None<br><br>for number in number s:<br>if number % 2 == 0:<br>found = number break<br>if found != None :<br>print("El primer número par es:", found)<br>else:<br>print("No hay números pares en la tupla") | number s = (19, 12, 35, 21, 12, 41, 37)<br>index = 0<br>found = None<br><br>while index < len( number s):<br><br>number = number s[index]<br>if number % 2 == 0:<br>found = number break<br><br>index += 1 |

```
if found != None :
print("El primer número par es:", found)
else:
print("No hay números pares en la tupla")
```

El ejemplo sigue un **esquema de búsqueda**, que es similar a un esquema de selección en cuanto que incluye una sentencia *if* para comprobar si el elemento actual de la secuencia cumple una condición, pero diferenciándose en que cuando ello ocurre se ejecuta la instrucción ***break***, la cual hace que se interrumpa la ejecución normal del bucle, dándolo por finalizado (la iteración actual termina, y ya no se realizan nuevas iteraciones).






Un esquema de búsqueda avanza mientras queden elementos que examinar y no se haya encontrado ninguno que cumpla la condición de búsqueda. Por tanto, el bucle puede terminar, bien porque se acabe la secuencia sin encontrar ningún elemento que cumpla la condición de búsqueda, bien porque se haya encontrado un elemento que la cumpla. Si después del bucle es necesario saber si se ha encontrado o no, habrá que usar una sentencia *if* para discernirlo.

Cuando se usa un *while*, el esquema de búsqueda puede evitar el *if* y el *break* usando una condición doble en el *while*:

```
numbers = (19, 13, 35, 21, 13, 41, 37)
index = 0
```

```
while index < len(numbers) andnumbers[index] % 2 != 0:
    index += 1
```

```
if index < len(numbers):
    print("El primer número par es:",numbers[index])
else:
    print("No hay números pares en la tupla")
```

Como puede observarse, en este caso solo la variable *index* nos da la pista para saber si se ha encontrado un elemento que cumple la condición, y cuál es en ese caso. De hecho, a veces

más que el valor del elemento en sí lo que nos interesa es su posición en la secuencia.

En todos los ejemplos hemos supuesto que solo nos interesa el primer elemento que cumple la condición, independientemente de que después de él haya otro(s) que también la cumplan.

## Alteración del flujo de un bucle

Existen dos instrucciones que permiten alterar el flujo normal de ejecución de un bucle, ambas aplicables a cualquier tipo de bucle, sea *while* o *for*. La instrucción ***break*** interrumpe la ejecución del bucle terminándolo completamente de forma anticipada. El siguiente ejemplo muestra cada elemento de la tupla *numbers* emparejado con su sucesor (en la sucesión de los números enteros), hasta que se encuentre el primer elemento par, en cuyo caso en lugar del sucesor se muestra la palabra *end* y se interrumpe el bucle, no realizándose el resto de la iteración actual (que hubiera mostrado el sucesor) ni tratándose los restantes elementos de la tupla.

```
numbers = (19, 21, 35, 21, 12, 41, 28, 37)

for element innumbers:
    print(element, end = "-")

    if element % 2 == 0:
        print("end")
        break

    print(element + 1)
```

El resultado es:

19-20

21-22

35-36

21-22

12-end

La instrucción ***continue***, por el contrario, termina la iteración actual, sin ejecutar las instrucciones que falten para esta, pero no termina el bucle, sino que pasa a intentar la iteración siguiente (lo mismo que si la iteración actual hubiese concluido normalmente).

```
numbers = (19, 21, 35, 21, 12, 41, 37)

for element innumbers:
    print(element, end = "-")

    if element % 2 == 0:
        print()
```

```
        continue

    print(element + 1)
```

El resultado del ejemplo anterior es:

19-20

21-22

35-36

21-22

12-

41-42

37-38

Podemos resumir diciendo que *continue* salta al principio del bucle mientras que *break* salta fuera del bucle.

# Depuración de programas con pdb

## Depuración

Al escribir un programa se pueden cometer errores sintácticos, esto es, errores de escritura que impiden que el programa pueda siquiera ejecutarse. Un error sintáctico es, por ejemplo, no poner dos puntos al final de la cabecera de una función.

Los errores sintácticos son fáciles de arreglar, porque sólo hay que cumplir las reglas del lenguaje. Una vez que un programa está correctamente escrito de acuerdo con dichas reglas, ya se puede ejecutar, pero durante la ejecución pueden aparecer errores de funcionamiento, originados ya sea por una equivocada concepción del programa o por no cumplir alguna regla del lenguaje que solo es factible comprobar en ejecución.

Es habitual que un programa mínimamente complejo funcione bien la mayoría de las veces, pero que, para unos pocos casos, falle dando resultados erróneos, o incluso, aborte (se detenga bruscamente sin terminar su tarea).

El ciclo de desarrollo de un programa pasa por diferentes fases. Las primeras son: análisis de los requerimientos y características del problema a resolver, diseño de la solución y codificación de la misma en un lenguaje de programación. Una vez que disponemos del programa ya ejecutable, se pasa a la fase de prueba, en la que se somete al programa a una batería de casos de prueba (*tests*) con el fin de encontrar casos de fallo a corregir en el código del programa.

Si las pruebas detectan un caso para el que el programa falla, se pasa a la fase de depuración que consiste en analizar el código para buscar, normalmente con ayuda de herramientas específicas, cuál es la causa del fallo: la **depuración** (*debugging*) es el proceso de buscar y solucionar defectos en el código de un programa que provocan que funcione incorrectamente.

Una herramienta de depuración (depurador o *debugger*) tiene que ofrecer funcionalidades para detener y reanudar la ejecución de un programa y, cuando está detenido, examinar el estado del proceso que lleva a cabo, inspeccionando los valores de las variables involucradas, la secuencia de llamadas a funciones realizada, el contexto de la instrucción actual, etc.

El depurador proporciona datos sobre el funcionamiento del programa que el programador tiene que analizar, razonando sobre ellos, para encontrar las causas del mal funcionamiento detectado por las pruebas.

none')?'inline':'none'})()" style="color:darkblue">Etimología

## Iniciando la depuración en Python

Python dispone de un módulo, llamado *pdb* (**p**ython **d**e**b**ugger), que sirve como herramienta de depuración. Para iniciar la depuración de un programa, sólo hay que importar ese módulo y ejecutar la instrucción *set_trace()*, tal como se hace en las dos primeras líneas del siguiente ejemplo:

```
import pdb
pdb.set_trace()
importrandom
def find(numbers, searched):
    found = False
    fornumber innumbers:
        ifnumber == searched:
            found = True
            break
    return found
if__name__ == '__main__':
numbers = (19, 12, 35, 21, 12, 41, 37)
number =random.randint(10, 50)

    if find(numbers,number):
        print("El número",number, "está en ",numbers)
```

1. **import pdb**
2. **pdb.set_trace()**
3. import random
4. def find(numbers, searched):
5. found = False
6. for number in numbers:
7. if number == searched:
8. found = True

9. break
10. return found
11. 
12. if __name__ == '__main__':
13. numbers = (19, 12, 35, 21, 12, 41, 37)
14. number = random.randint(10, 50)
15. 
16. if find(numbers, number):
17. print("El número", number, "está en ", numbers)

Esta instrucción hace que se arranque el depurador y se quede esperando órdenes para continuar la ejecución del programa, que queda detenido justo en la línea siguiente:

```
> /home/p16251/main.py(3)<module>()
→ importrandom
(Pdb)
```

Obsérvese que justo antes del *prompt* del depurador (Pdb) hay una flecha que indica que el programa está detenido en la línea cuyo contenido es "import random", la línea 3 de nuestro programa de ejemplo. Un comando muy útil es este momento, es *help*, que nos mostraría todos los comandos del depurador:

```
> /home/p13011/main.py(3)<module>()
→ importrandom
(Pdb) help
Documented commands (type help <topic>):
========================================
EOF    c          d          h         list      q        rv       undisplay
a      cl         debug      help      ll        quit     s        unt
alias  clear      disable    ignore    longlist  r        source   until
args   commands   display    interact  n         restart  step     up
b      condition  down       j         next      return   tbreak   w
break  cont       enable     jump      p         retval   u        whatis
bt     continue   exit       l         pp        run      unalias  where
Miscellaneous help topics:
==========================
exec  pdb
(Pdb)
```

*help* seguido de un comando concreto, nos da ayuda sobre ese comando en particular.

## Breakpoints (puntos de parada)

Una cosa que podemos hacer cuando iniciamos la depuración es establecer **puntos de parada** (*breakpoints*). De esta manera, podemos hacer que la ejecución se vaya deteniendo en distintos puntos del programa, permitiéndonos inspeccionar el estado del proceso en cada momento. En el siguiente ejemplo se ponen dos puntos de parada en las líneas 6 y 14 de nuestro programa de ejemplo:

```
> /home/p17284/main.py(3)<module>()
→ importrandom
(Pdb) b 6
Breakpoint 1 at /home/p17284/main.py:6
(Pdb) b 14
Breakpoint 2 at /home/p17284/main.py:14
(Pdb)
```

Para poner un punto de parada se puede usar el comando *break*, o su forma abreviada, *b*, como en el ejemplo.

## Avance

Cuando un programa está detenido en un punto de parada, o al inicio de la depuración, se puede **reanudar la ejecución** (hasta el siguiente punto de parada, si existe), usando el comando ***continue***, o sus versiones abreviadas, *cont* o ***c***:

```
> /home/p14373/main.py(3)<module>()
→ importrandom
(Pdb) b 14
Breakpoint 1 at /home/p14373/main.py:15
(Pdb) c
> /home/p14373/main.py(14)<module>()
→numbers = (19, 12, 35, 21, 12, 41, 37)
(Pdb)
```

Nótese que, siempre, antes del prompt del depurador, una flecha nos muestra la línea en la que el programa está detenido, línea que aún no se ha ejecutado (justo antes, se muestra también el número de dicha línea).

Una opción alternativa es **avanzar solo una línea**, ejecutando la actual y parando antes de la siguiente, con el comando ***next***, o ***n***:

```
> /home/p16883/main.py(14)<module>()
→numbers = (19, 12, 35, 21, 12, 41, 37)
(Pdb) n
> /home/p16883/main.py(15)<module>()
→number =random.randint(10, 50)
(Pdb) n
> /home/p16883/main.py(17)<module>()
```

```
→ if find(numbers,number):
(Pdb)
```

Cuando la próxima instrucción a ejecutar contenga una llamada a una función, cabe la posibilidad de usar el comando *next*, que ejecuta la llamada a la función y se para en la siguiente instrucción después de que la función retorne, o el comando ***step***, abreviado ***s***, que se para en la primera instrucción de la función, permitiéndonos continuar la depuración por dentro de la misma:

```
> /home/p17498/main.py(17)<module>()
→ if find(numbers,number):
(Pdb) s
--Call--
> /home/p17498/main.py(5)find()
→ def find(numbers, searched):
(Pdb)
```

## Inspección

Cuando el programa está detenido, podemos inspeccionar el estado del proceso, para averiguar si va bien o ya se ha desviado de la situación esperada, lo que significa que el fallo del programa está antes del punto actual. Una de las cosas que podemos ver es el **contexto de la línea actual**, usando el comando ***list*** o ***l***:

```
> /home/p12875/main.py(17)<module>()
→ if find(numbers,number):
(Pdb) l
 12
 13  if__name__ == '__main__':
 14numbers = (19, 12, 35, 21, 12, 41, 37)
 15number =random.randint(10, 50)
 16
 17 B→     if find(numbers,number):
 18           print("El número",number, "está en ",numbers)
 19       else:
 20           print("El número",number, "no está en ",numbers)
(Pdb)
```

Para **inspeccionar el valor de una variable** usamos el comando ***p*** (***print***):

```
> /home/p12608/main.py(17)<module>()
→ if find(numbers,number):
(Pdb) pnumbers
(19, 12, 35, 21, 12, 41, 37)
(Pdb) pnumber
```

```
40
(Pdb)
```

En el ejemplo vemos que *numbers* referencia una lista con siete valores enteros y que *number* referencia el valor entero 40.

El comando ***where*** (abreviado ***w***) permite conocer la **secuencia de llamadas** que nos han conducido al punto actual. Esto es útil cuando a una función se la puede llamar desde diferentes puntos de un programa, ya que el camino que se ha seguido para llegar al punto actual puede ser relevante para determinar cómo se produce el fallo:

```
> /home/p15071/main.py(6)find()
-> found = False
(Pdb) w
  /home/p15071/main.py(17)<module>()
-> if find(numbers,number):
> /home/p15071/main.py(6)find()
-> found = False
(Pdb)
```

En el ejemplo, vemos que el programa está detenido en la función *find()*, en la línea 6 del módulo *main.py*, adonde se ha llegado ejecutando la llamada a *find()* que se encuentra en la línea 17 del mismo módulo.

# Listas

Las **listas** son *estructuras de datos* diseñadas para almacenar secuencias de elementos.

Para crear una lista en Python, ponemos los elementos separados por comas y encerrados entre corchetes:

```
temperatures = [36.5, 36.7, 37.1, 37.1, 37.5, 38.5]
students = ['Ada Lovelace', 'Alan Turing', Edsger W. Dijkstra', 'Donald Knuth', 'Niklaus Wirt
h', 'Ole-Johan Dahl', 'Kristen Nygaard']
years = [1945, 1969, 1971, 1978, 1980, 1984, 2012, 2015, 2019]
incoming_person = ['Susan', 'Slovakia’, 2019, 52.18, True]
x = 17
y = 9
z = 1989
values = [x, y, z]
```

Como se ve en el ejemplo, en Python una lista puede tener todos los elementos del mismo tipo, o mezclar elementos de tipos diferentes.

Una lista vacía se puede crear de dos formas.

```
a = []
```

o

```
a = list()
```

La función type() devuelve el tipo list cuando se le pasa una lista:

```
>>> type(a)
<class 'list'>
```

Para acceder a los elementos de una lista se usan índices. El índice del primer elemento es 0. Igual que en las strings o las tuplas, se pueden usar índices negativos que referencian desde el final de la lista (el índice negativo del último elemento es -1).

```
a = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
print(a[0])
print(a[2])
print(a[-1])
print(a[-2])
print(a[len(a)-1])
```

Resultado de la ejecución del código anterior:

Monday

Wednesday

Sunday

Saturday

Sunday

## Concatenación de listas

Se pueden concatenar dos listas usando el operador +. Como resultado creamos una nueva lista conteniendo todos los elementos de la primera lista, seguidos de todos los elementos de la segunda lista, conservando siempre el orden original.

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> b = [6, 7, 8]
>>> a + b
[1, 2, 3, 4, 5, 6, 6, 7, 8]
>>> students = ['John', 'Martin']
>>> students =  ['Joseph'] + students
>>> students
```

```
['Joseph', 'John', 'Martin']
>>> students + []
['Joseph', 'John', 'Martin']
```

### El operador *

El operador * se usa para concatenar una misma lista repetidamente.

```
>>> languages = ['C', 'Java', 'Python'] + ['C', 'Java', 'Python'] + ['C', 'Java', 'Python']
>>> languages
['C', 'Java', 'Python', 'C', 'Java', 'Python', 'C', 'Java', 'Python']
>>> languages = ['C', 'Java', 'Python'] * 3
>>> languages
['C', 'Java', 'Python', 'C', 'Java', 'Python', 'C', 'Java', 'Python']
```

El operador * facilita la inicialización de una lista:

```
>>> counters = [0] * 10
>>> counters
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

### Operadores in y not in

Podemos comprobar si un elemento determinado está o no contenido en una lista usando los operadores in y not in:

```
>>> languages = ['C', 'Java', 'Python']
>>> 'Pascal'inlanguages
False
>>> 'Python'inlanguages
True
>>> 'Java'not in languages
False
```

### Anotación de variables lista

A partir de la versión 3.9 de Python se puede *anotar* usando el nombre real (en minúsculas) de tipos no básicos:

```
lista: list        # Lavariable lista se anota de tipo list
vector: list[float] # Lavariable vector se anota como una lista de elementos de tipo float
```

En versiones anteriores es necesario usar un nombre del módulo **typing**:

```
from typing import List
lista: List
vector: List[float]
```

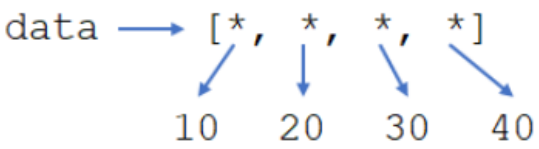
Recordemos que Python por sí mismo **no impide** que a una variable se les asignen valores de un tipo distinto del anotado.

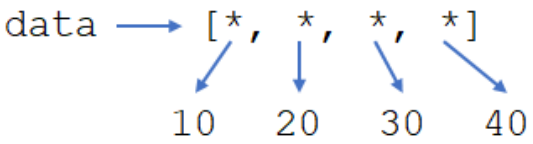
## Las listas son mutables

Considere la siguiente instrucción:

```
data = [10, 20, 30, 40]
```

Esta instrucción crea una lista referencia da por la variable *data*.



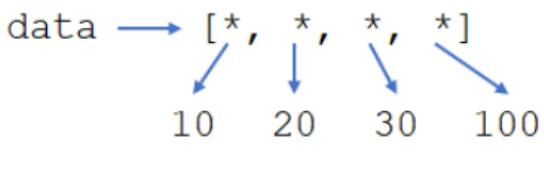

La lista consta de 4 variables: data[0], data[1], data[2] y data[3] que hacen referencia a los cuatro valores *int*.

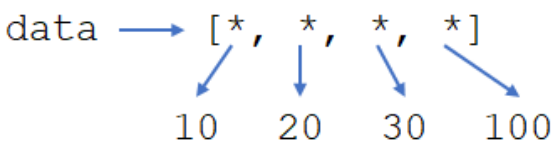
Las listas son estructuras mutables, esto significa que pueden ser modificadas.

```
data[3] = 100
```

La instrucción anterior cambia el último elemento de la lista.

No hemos creado una nueva lista, solo hemos cambiado uno de sus elementos, de ahí que digamos que sea mutable.



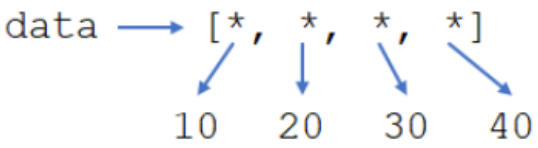


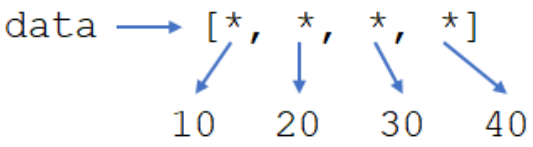

Nótese que esto no se puede hacer con una tupla o con una string porque, a diferencia de las listas, ambas son estructuras de datos inmutables.

## Eliminación de elementos

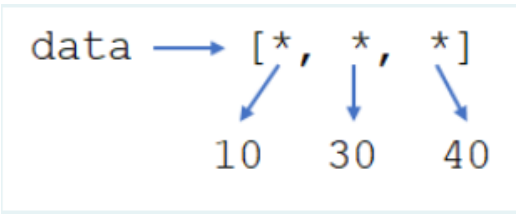
Para eliminar un elemento de una lista se puede usar la instrucción **del**.

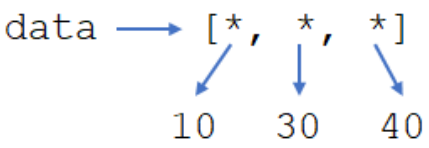
```
data = [10, 20, 30, 40]
```




```
del data[1]
```




## Recorrido de una lista

Para recorrer una lista suele usarse un bucle *for*:

```
friends = ['Peter','Paul', 'Michael', 'George', 'John']
foriinrange(len(friends)):
    print(i, "-", friends[i])
```

Resultado:

0 - Peter

1 - Paul

2 - Michael

3 - George

4 - John

Si sólo estamos interesados en los elementos y los índices no son relevantes, podemos usar una notación más concisa:

```
forelementinfriends:
    print(element, end=' ')
```

En este último ejemplo, además, hemos añadido a la instrucción *print* el parámetro *end*, sustituyendo el salto de línea que se usa por omisión por un espacio; el resultado es:

Peter Paul Michael George John

## Enumerate

Podemos hacer uso de la función **enumerate**() para acceder a ambos, el índice y el valor correspondiente al mismo tiempo:

```
friends = ['Peter', 'Paul', 'Michael', 'George', 'John']

for i, element in enumerate(friends):
   print(i, "-", element)
```

Resultado:

0 - Peter

1 - Paul

2 - Michael

3 - George

4 - John

La función *enumerate()* inicia el contador de la secuencia de posiciones en 0; podemos especificar el valor inicial usando el parámetro *start*:

```
friends = ['Peter', 'Paul', 'Michael', 'George', 'John']

fori, elementinenumerate(friends, start = 1):   print(i, "-", element)
```

Resultado:

1 - Peter

2 - Paul

3 - Michael

4 - George

5 - John

## ***List comprehension***

La *comprensión de listas* de Python es un mecanismo que proporciona una forma concisa de crear nuevas listas. Consiste en una expresión, seguida de una cláusula for, seguida a su vez de cero o más cláusulas for o if; todo ello entre corchetes. Por ejemplo:

```
[expression forelement inlist]
[expression forelement inlist ifcondition]
```

Con esta notación obtenemos una nueva lista, compuesta de los valores que resulten de evaluar repetidamente la expresión (que puede ser compleja) en el contexto de las cláusulas for e if que la siguen. Aquí hay algunos ejemplos; estudie atentamente los resultados:

```
list1 =  [iforiinrange(10)]
#list1 vale: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
list2 = [i ** 2foriinrange(5)]
#list2 vale: [0, 1, 4, 9, 16]
```

none')?'inline':'none'})()" style="color:darkblue">Etimología

## List comprehension - más ejemplos

```
import random
a = [random.randint(0, 9)for i in range(10)]
print(a)
```

*Salida: [4, 4, 9, 7, 4, 1, 0, 0, 3, 6]*

(La función *random.randint()* genera un número al azar en el rango indicado; la salida será diferente en diferentes ejecuciones)

```
words = ['this', 'is', 'a', 'list', 'of', 'words']
b = [word[0].upper()for word in words]
print(b)
```

*Salida: ['T', 'I', 'A', 'L', 'O', 'W']*

(El método *upper()* devuelve la conversión a mayúsculas de la string asociada)

```
things = ['pen', 'pencil', 'rubber', 'paper']
c = [thing for thing in things if len(thing) > 5]
print(c)
```

*Salida: ['pencil', 'rubber']*

```
from math import pi
d = [str(round(pi, i))for i in range(1, 6)]
print(d)
```

*Salida: ['3.1', '3.14', '3.142', '3.1416', '3.14159']*

(La función *round()* redondea el *float* pasado como primer parámetro al número de decimales indicado por el segundo)

# Funciones de manejo de listas

Python ofrece algunas funciones estándar útiles para manejar listas, por ejemplo:

| | |
|---|---|
| **len**(*list*) | devuelve el número de elementos de una lista (su longitud) |
| **sum**(*list*) | devuelve la suma de los elementos de una lista |
| **max**(*list*) | devuelve el valor máximo de los elementos de una lista |
| **min**(*list*) | devuelve el valor mínimo de los elementos de una lista |

Lógicamente, sum() sólo se puede aplicar a listas con elementos numéricos, y max() y min() a listas con valor es comparables entre sí.

## Función list()

La función **list**() crea una lista con los elementos de una secuencia.

```
a = list(range(1, 10))  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
b = list('Python')      # ['P', 'y', 't', 'h', 'o', 'n']
c = list()              # []
d = ('milk', 'bread', 'butter','honey')  # tupla
e = list(d)             # ['milk', 'bread', 'butter', 'honey']
f = list(a)
```

En el último ejemplo, f recibe una **nueva** lista, copiada de la lista a.

## Métodos de manejo de listas sin modificación

En Python los objetos de tipo list también admiten algunos *métodos*, que son funciones especiales que han de ser llamadas indicando el objeto sobre el que se aplican, seguido de un punto: por ejemplo, dada la lista mylist, los siguientes dos métodos proporcionan información sobre la misma (sin cambiar su contenido):

```
a = mylist.count(x)            # Devuelve el número de veces que x aparece en mylist
b = mylist.index(x)            # Devuelve la posición de la primeraocurrencia de x en mylist
c = mylist.index(x, start, end)  # Devuelve la posición de la primeraocurrencia de x en mylis
t
                          # en el rango deíndices start y end
```

Ejemplo:

```
fruits = ['banana', 'apple', 'pear', 'cherry', 'banana', 'banana']
print(fruits.count('banana'))               # 3
print(fruits.index('banana'))               # 0
print(fruits.index('banana', 3, len(fruits)))  # 4
```

Debemos tener cuidado al usar el método index, ya que si el valor buscado no se encuentra en la lista se lanza la excepción **ValueError**. Podemos prevenirlo usando el operador *in*:

```
if'peach' infruits:
   print(fruits.index('peach'))
else:
   print('There is no "peach" in this list.')
```

## Métodos que modifican listas

A los siguientes métodos se les llama (del inglés) *"mutables"*, pues modifican el contenido de la lista sobre la que se aplican (no crean una nueva lista).

Sea mylist una lista cualquiera:

```
mylist.append(x)         # Añade un nuevo elemento con elvalor x al final de mylist
mylist.insert(i, x)      # Inserta un nuevo elemento con elvalor x en la posición de mylist indi
cada por i
value = mylist.pop()     # Quita de mylist su último elemento y lo devuelve; se asigna a lava
riable value
value = mylist.pop(i)    # Quita de mylist el elemento indicado por i y lo devuelve; se asigna
a lavariable value
mylist.remove(x)         # Quita de mylist el primer elemento cuyovalor sea igual a x
```

```
mylist.sort()            # Ordena mylist de menor a mayorvalor
mylist.sort(reverse=True) # Orena mylist de mayor a menorvalor
mylist.sort(key=getKey)   # Siendo getKey el nombre de unafunción que toma como parám
eto un elemento
                          # Ordena la lista enfunción de los resultados de getKey al aplicarse a cada
                          # elemento de la lista. Puede combinarse con reverse.
mylist.extend(other)      # Añade al final de mylist los elementos de la secuencia other
mylist.clear()            # Vacía mylist, quitando todos los elementos
```

Tanto pop() como remove() lanzan la excepción ValueError si no existe elemento a extraer. Si el índice de insert es mayor o igual que la longitud de la lista, añade al final.

Existe una función:

```
sorted(c)→ list sorted copy
```

que, siendo c una secuencia/contenedor, devuelve una lista ordenada con los elementos de c; por ejemplo:

```
c = "cabrito"
s = sorted(c)
print(s)  # ['a', 'b', 'c', 'i', 'o', 'r', 't']
```

La función sorted() admite los mismos parámetros opcionales que el método .sort() -reverse y key.

## Métodos que modifican listas

Ejemplo de código:

```
fruits = ['banana', 'apple', 'pear', 'cherry', 'apple', 'banana']
fruits.append('peach')
print(fruits)            # ['banana', 'apple', 'pear', 'cherry', 'apple', 'banana', 'peach']
print(fruits.pop())       # peach
print(fruits.pop(0))      # banana
fruits.insert(0, 'coconut')
fruits.remove('apple')
print(fruits)            # ['coconut', 'pear', 'cherry', 'apple', 'banana']
fruits.reverse()
print(fruits)            # ['banana', 'apple', 'cherry', 'pear', 'coconut]
fruits.sort()
print(fruits)            # ['apple', 'banana', 'cherry', 'coconut', 'pear']

def lastChar(string):      # Obtiene el último carácter de unastring
   returnstring[-1]
```

```
fruits.sort(key=lastChar, reverse=True)  # Ordena por el último caracter, descendente
print(fruits)            # ['cherry', 'coconut', 'pear', 'apple', 'banana']
fruits.extend(['orange', 'lemmon'])
print(fruits)            # ['cherry', 'coconut', 'pear', 'apple', 'banana', 'orange', 'lemmon']
fruits.clear()
print(fruits)            # []
```

En el siguiente ejemplo se añaden elementos a una lista vacía para que contenga todos los números enteros no negativos menores que 1000 divisibles por 3 o por 5 o por ambos:

```
numbers = []
foriinrange(1000):
ifi % 3 == 0ori % 5 == 0:
numbers.append(i)
```

### *Slices*

Igual que con las strings o las tuplas, podemos obtener sublistas (*slices*, segmentos) a partir de una lista. La sintaxis completa es:

```
lista[inicio : fin : paso]
```

La operación de slice no modifica la lista original, sino que crea una nueva.

Ejemplo de código:

```
cities = ['Bratislava', 'Warsaw', 'Madrid', 'Praha']
a = cities[1:3]   # ['Warsaw', 'Madrid']
b = cities[-3:]   # ['Warsaw','Madrid', 'Praha']
c = cities[:-1]   # ['Bratislava','Warsaw', 'Madrid']
d = cities[1::2]  # ['Warsaw', 'Praha']
e = cities[::-1]  # ['Praha', 'Madrid', 'Warsaw', 'Bratislava']
f = cities[:]     # crea unanueva lista ['Bratislava', 'Warsaw', 'Madrid', 'Praha']
```

## Asignación a slices

Cuando se toma un slice de una lista siempre se crea una lista nueva sin modificar la original.

La notación de slices puede usarse también a la izquierda de una asignación (como destino de la asignación); en ese caso, la parte derecha de la asignación (el origen de la asignación) debe ser una expresión que represente una secuencia (no obligatoriamente una lista). El funcionamiento de la asignación es como sigue:

1. Se evalúa la parte derecha de la asignación creando una lista nueva.

2. La nueva lista sustituye a la sublista representada por el slice, modificando la lista a la que el slice hace referencia.

### Ejemplos en secuencia de instrucciones:

1) Se sustituyen tres elementos por otros tres.

```
names = ['Matthew', 'Mark', 'Luke', 'John', 'Francis']
names[1:4] = ['Paul', 'Peter', 'Thomas'] # ['Matthew','Paul', 'Peter', 'Thomas', 'Francis']
```

2) Tres elementos se sustituyen por dos.

```
names = ['Matthew', 'Mark', 'Luke', 'John', 'Francis']
names[1:4] = ['Paul', 'Peter'] # ['Matthew', 'Paul', 'Peter', 'Francis']
```

3) Los dos últimos elementos son sustituidos por tres.

```
names[-2:] = ['Andrew', 'James','Philip'] # ['Matthew', 'Paul', 'Andrew', 'James', 'Philip']
```

4) Los dos primeros elementos son sustituidos por una lista vacía.

```
names[:2] = [] # ['Andrew', 'James', 'Philip']
```

5) El primer elemento se sustituye por seis elementos de una lista construida a partir de una string.

```
names[:1] = 'Python' # ['P', 'y', 't', 'h', 'o', 'n', 'James', 'Philip']
```

## Comparación de listas

Se pueden comparar dos listas usando los operadores relacionales:

- Se van comparando de principio a fin los elementos que ocupan las mismas posiciones en las dos listas hasta que se encuentra una pareja diferente o se acaba alguna de las listas.
- Si se encuentra una pareja de elementos diferentes, es menor la lista que tenga el elemento menor.
- Si se acaba una de las listas, sin encontrar una pareja diferente, y la otra no, es menor la lista de menor longitud.
- Si se llega a la vez al final de las dos listas sin encontrar una pareja de elementos diferentes, las listas son iguales.

Ejemplos en secuencia de instrucciones:

```
a = list(range(1,5))   #  [1, 2, 3, 4]
b = list(range(1,5))   #  [1, 2, 3, 4]
a == b     # True
a != b     # False
a[1] = 0
a == b     # [1, 0, 3, 4] == [1, 2, 3, 4] da False
a < b      # [1, 0, 3, 4] < [1, 2, 3, 4] da True
```

En la lista *b*, el elemento con índice 1 es mayor que el elemento correspondiente de la lista *a*.

```
a <= b     # True
a > b      # False
a >= b     # False
[1, 2] < b  # [1, 2] < [1, 2, 3, 4] da True
```

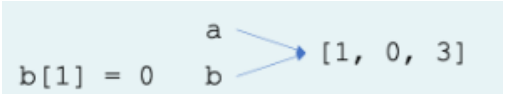
Los primeros dos elementos en las dos listas son iguales, pero la primera lista es más corta.

# Listas y funciones

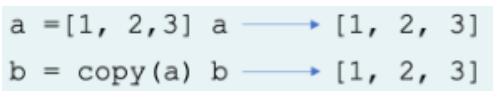
## Alias

Una lista puede ser referenciada por más de una variable:


Las variables a y b referencian la misma lista. Decimos que ambas variables son *alias* del mismo objeto (una lista, en este caso). Esto significa que si se modifica el objeto referenciado por una de las variables (la lista en cuestión), el objeto referenciado por la otra (que es el mismo) lo veríamos asimismo modificado. En el siguiente ejemplo, tras asignar b[1]=0, veríamos que el elemento a[1] pasaría a valer 0, y no 2, como originalmente.

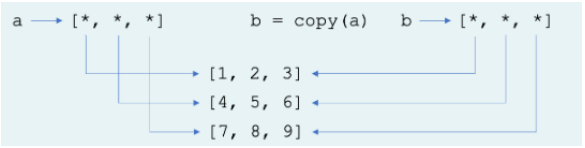



Si se necesita tener una copia independiente, se puede usar la función copy(), del módulo copy, que hay que importar. Como resultados tendremos dos listas que, aunque de momento son iguales, no son la misma lista (no son el mismo objeto). Por tanto, ahora la asignación b[1]=0 no tendría efecto sobre a[1], que seguiría valiendo 2.

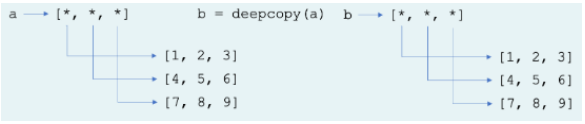



### *Shallow* copy y *deep* copy

La función copy() del módulo copy realiza una copia "superficial" o "somera". Esto significa que si los elementos de la lista, en vez de referenciar valores de un tipo simple, como números enteros, son a su vez referencias a objetos (p.e., una lista de listas), los elementos de 2º nivel (y sucesivos niveles, si los hubiera) quedan compartidos, ya que sólo se copian las referencias de 1er nivel:

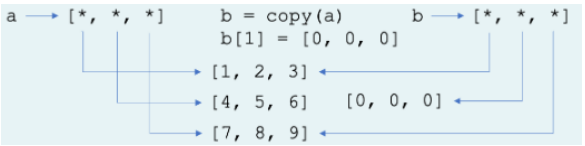



Si asignamos a[0][1]=5, el valor al que accederíamos a continuación mediante b[0][1] lo veríamos también modificado, ya que a[0] y b[0] referencian el mismo objeto lista.




Si se quiere hacer una copia "profunda", hay que usar la función deepcopy() en vez de la copy():

De todas formas, la compartición de elementos de segundo nivel y no del primero, permite modificar elementos del primer nivel sin que se modifiquen los correspondientes del segundo de la otra variable:




## Paso de listas como parámetros

Cuando tenemos que procesar una lista, frecuentemente la pasamos como parámetro a una función. En estos casos el parámetro formal de la función referenciará a la misma lista que pasamos en la llamada.

La función del siguiente ejemplo devuelve el valor máximo almacenado en una lista, de hecho, en cualquier secuencia iterable (como por ejemplo una tupla o una ristra). Podemos además ver que para esta función ejemplo los elementos de la secuencia iterable deben ser comparables entre sí, ya que no funcionaría si mezclamos datos de p.e. tipos int y str:

```
defmaximum(a):
    m = a[0]
foriinrange(1, len(a)):
ifa[i] > m:
            m = a[i]
returnm
```

```
data = [1, 3, 8, 5, 6]
print(maximum(data))      # 8
print(data)               # [1, 3, 8, 5, 6]
print(maximum('Python'))  # y
```

Obsérvese que en el último caso hemos usado *maximum* con una *str*, secuencia iterable compuesta de caracteres.

En el ejemplo anterior no hemos modificado la lista recibida por parámetro. Por el contrario, en el ejemplo siguiente la función intercambia dos elementos de una lista (se supone que tanto *i* como *j* son índices válidos para la lista *a*):

```
defexchange(a, i, j):
   a[i], a[j] = a[j], a[i]

values = [2, 6, 8, 5, 6, 9]
exchange(values, 1, 3)
print(values)            # [2, 5, 8, 6, 6, 9]
```

En ambos ejemplos, el parámetro formal, *a*, es una referencia al parámetro real que se pasa en la llamada, que en el segundo ejemplo es *values*, por lo que, al modificar la lista usando *a* en la función, se está modificando la misma lista referencia da por *values* desde fuera de la función. En este caso a*exchange*, a diferencia de *maximum*, solo se le puede pasar una secuencia mutable.

## Listas como resultado de funciones

Además de poder pasar referencias a listas como parámetros, las referencias a listas también pueden devolverse como resultado de una función:

```
defcreate_list(n, value = 0):
return[value] * n

print(create_list(10))       # [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
print(create_list(5, -1))    # [-1, -1, -1, -1, -1]
print(create_list(8,None))   # [None,None,None,None,None,None,None,None]
print(create_list(5, 'Hi'))  # ['Hi', 'Hi', 'Hi', 'Hi', 'Hi']
```

En tanto en cuanto no especifiquemos el tipo del resultado esperado y no usemos una herramienta para tenerlo en cuenta, la función *create_list()* es una plantilla general para crear listas de elementos con un valor determinado.

Para especificar que se requiere una lista como parámetro o se espera como resultado podemos usar, igual que para los tipos básicos, el nombre de tipo *list*. Si quisiéramos especificar además de qué tipo deben ser, a su vez, los elementos de la lista, podemos importar del módulo *typing* el nombre *List*, como en el siguiente ejemplo:

```
from typing import List

def create_list(n: int, value: int = 0) → List[int]:
    return [value] * n
```

Para las tuplas (tipos *tuple* y *typing.Tuple*) ocurre igual.

### Funciones modificadoras y no modificadoras

Cuando escribimos funciones que manejan listas, debemos tener presente si queremos que la lista original se modifique o no.

```
defadd1(list, element):
returnlist + [element]
defadd2(list, element):
    list.append(element)
returnlist

data = [1, 3, 8, 5, 6]
print(add1(data, 1000))         # [1, 3, 8, 5, 6, 1000]
print(data)                     # [1, 3, 8, 5, 6]
print(add2(data, 1000))         # [1, 3, 8, 5, 6, 1000]
print(data)                     # [1, 3, 8, 5, 6, 1000]
```

En el ejemplo anterior, la función *add1()* devuelve una referencia a una nueva lista, sin modificar la original, mientras que la función *add2()* añade un elemento a la lista original y devuelve la referencia a la lista original ya modificada. Decimos que la función *add1()* es no modificadora y la función *add2()* es modificadora.

# Tratando múltiples listas a la vez

### Procesando varias listas al mismo tiempo

En muchas ocasiones se necesita procesar dos o más listas simultáneamente. El siguiente ejemplo intercala los elementos de dos listas, generando una nueva:

```
def interleave(list1, list2):
    result = []

    for i in range(len(list1)):
        result.append(list1[i])
        result.append(list2[i])

    return result
```

```
numbers = [10, 20, 30]
chars= ['A', 'B', 'C']
print(interleave(numbers, chars)) # [10, 'A', 20, 'B', 30, 'C']
```

Como se observa en el ejemplo, no hace falta un bucle independiente para cada lista, ya que de lo que se trata es de recorrerlas al mismo tiempo, tomando un elemento de cada una para pasarlo al resultado, esto es, 1 recorrido (de 1, 2, o más listas a la vez) == 1 bucle.

En la función del ejemplo se supone que ambas listas son del mismo tamaño, o que *len(list2) >= len(list1)*. De no ser así, se produciría un error (*Index out of range*).

## Procesando varias listas al mismo tiempo con zip

Si no necesitamos tener acceso a los índices, una forma de procesar dos o más listas al mismo tiempo es usar la función *zip()*:

```
def interleave1(list1, list2):
    result = []

    foritem1,item2 in zip(list1, list2):
        result.append(item1)
        result.append(item2)

    return result

list1 = [10, 20, 30]
list2 = ['A', 'B', 'C']
print(interleave1(list1, list2)) # [10, 'A', 20, 'B', 30, 'C']
```

La función predefinida *zip()* permite iterar por una secuencia de tuplas, formada de tal manera que el *i*-ésimo elemento de esa secuencia es una tupla formada por los i-ésimos elementos de cada una de las listas (en general, secuencias) que se le pasan como parámetros. En el ejemplo, el primer elemento devuelto por *zip* es una tupla formada por el primer elemento de *list1* y el primer elemento de *list2 (10, 'A')* como segundo elemento *(20, 'B')* y como tercer y último elemento *(30, 'C')*. Note que en cada iteración del bucle, cada elemento/tupla de la secuencia zip es disgregado en las variables item1 e item2. Para la tupla *(10, 'A')* *item1* tomará el valor *10* e *item2* el valor *'A'*.

En el caso de que las listas fuesen de diferentes longitudes, *zip()* procesaría sólo hasta la longitud de la más corta.

## Procesando simultáneamente listas de distintos tamaños con zip_longest

La función *zip()* permite recorrer simultáneamente dos o más listas, pero sólo hasta el tamaño de la más corta. Si se quiere recorrer hasta el tamaño de la más larga, se puede usar la función *zip_longest()* del módulo *itertools*:

```
import itertools
def interleave2(list1, list2):
    result = []

    foritem1,item2 in itertools.zip_longest(list1, list2, fillvalue='-'):
        result.append(item1)
        result.append(item2)

    return result

list1 = [10, 20, 30]
list2 = ['A', 'B', 'C', 'D']
print(interleave2(list1, list2)) # [10, 'A', 20, 'B', 30, 'C', '-', 'D']
```

Los elementos que faltan en la lista más cortas se sustituyen con el valor especificado en el parámetro *fillvalue* (el caracter '-' en el ejemplo). Si se omite el parámetro *fillvalue*, se usa el valor *None*.

## Procesando varias listas de diferentes tamaños a la vez usando índices

Si se quiere recorrer dos listas de diferentes tamaños a la vez usando índices en vez de *zip_longest()*, es conveniente, como en el siguiente ejemplo, separar la iteración en tres bucles de los que sólo se ejecutan, como máximo, dos: el primero, mientras las dos listas tienen elementos, y uno de los otros dos, para terminar de procesar los elementos de la lista más larga.

```
def interleave3(list1, list2):
    result = []
    min_len = min(len(list1), len(list2))

    for i in range(min_len):
        result.append(list1[i])
        result.append(list2[i])
```

```
    for i in range(min_len, len(list1)):
        result.append(list1[i])
        result.append('-')
```

```
    for i in range(min_len, len(list2)):
        result.append('-')
        result.append(list2[i])
```

```
    return result
```

Alternativamente, se pueden usar sentencias *while* en su lugar, aprovechando el valor del índice a la salida del primer bucle:

```
def interleave3b(list1, list2):
    result = []
    i = 0

    while i < min(len(list1), len(list2)):
        result.append(list1[i])
        result.append(list2[i])
        i += 1

    while i < len(list1):
        result.append(list1[i])
        result.append('-')
        i += 1

    while i < len(list2):
        result.append('-')
        result.append(list2[i])
        i += 1

    return result
```

De hacer un solo bucle, hasta la longitud más larga, habría que incluir dentro del bucle preguntas para saber si se ha acabado alguna de las listas, cuyo coste de ejecución se sumaría en cada ejecución, pero sería fácilmente extensible a cualquier número de listas:

```
def interleave3c(list1, list2):
    result = []

    for i in range(max(len(list1), len(list2))):
        if i < len(list1):
            result.append(list1[i])
        else:
            result.append('-')
        if i < len(list2):
            result.append(list2[i])
        else:
            result.append('-')

    return result
```

Una alternativa a esto, en Python, es usar *zip_longest()* junto con *enumerate()*, para dos o más listas:

```
from itertools import zip_longest

def interleave3d(list1, list2):
    result = []

    for i,items in enumerate(zip_longest(list1, list2, fillvalue='-')):
        result.append(items[0])
        result.append(items[1])

    return result
```

# Estructuras 2D

## Listas de tuplas

Las listas y las tuplas son tipos de datos estructurados que almacenan secuencias de elementos. Las listas son mutables y las tuplas son inmutables. Los elementos de una lista o de una tupla pueden ser de cualquier tipo, incluyendo (referencias a) tipos estructurados. Por ejemplo, podemos definir una lista de tuplas que representen coordenadas de puntos en el plano:

```
a = (0, 1)
b = (1, 1)
c = (2, -5)
d = (3, 7)
e = (4, 8)
points = [a, b, c, d, e]
print(points)           # [(0, 1), (1, 1), (2, -5), (3, 7), (4, 8)]
```

La variable *points* referencia a una lista que contiene 5 (referencias a) tuplas.

Podemos añadir elementos a la lista *points*, actualizarlos (sustituirlos), borrarlos (eliminarlos de la lista) o aplicar una operación de slice.

```
points.append((5, 0))
points.pop(0)
print(points)           # [(1, 1), (2, -5), (3, 7), (4, 8), (5, 0)]
points[2] = (1000, 1000)
print(points)           # [(1, 1), (2, -5), (1000, 1000), (4, 8), (5, 0)]
print(len(points))      # 5
alias = points
points[:] = points[::-1]
print(alias)            # [(5, 0), (4, 8), (1000, 1000), (2, -5), (1, 1)]
```

Podemos acceder a los elementos individuales de una tupla usando un segundo índice:

```
print(points[0])        # (5, 0) - la primera tupla de la lista
print(points[0][0])      # 5 - el primer elemento de la primera tupla
print(points[0][1])      # 0 – el segundo elemento de la primera tupla
```

Obviamente, dado que las tuplas son inmutables, la siguiente instrucción es errónea:

```
points[1][0] = 0
#... TypeError: 'tuple' object does not supportitem assignment
```

## Tablas como listas de listas

A menudo, necesitamos implementar una estructura bidimensional: una tabla con filas y columnas (una matriz de valores desde un punto de vista matemático). Dicha tabla de valores se puede crear utilizando listas de listas en Python:

```
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

o, en una forma más legible:

```
m = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

Cada uno de los elementos m[0], m[1], m[2] referencia a una lista de 3 elementos. Otra forma de crear la misma estructura:

```
row1 = [1, 2, 3]
row2 = [4, 5, 6]
row3 = [7, 8, 9]
m =[row1, row2, row3]
```

El número de filas o columnas se puede conocer usando la función *len()*.

```
print(len(m))     # 3 - número de filas
```

La longitud de cualquier elemento, por ejemplo m[0], nos da el número de columnas.

```
print(len(m[0]))   # 3 - número de columnas
```

Los elementos de la tabla se pueden acceder mediante una pareja de índices.
La expresión m[i][j] representa el valor del elemento que está en la columna j de la fila i. El siguiente ejemplo cambia el valor de la primera columna de la segunda fila de la tabla m y muestra la tabla:

```
m[1][0] = 0
print(m)          # [[1, 2, 3], [0, 5, 6], [7, 8, 9]]
```

## Recorrido de una tabla

Consideremos de nuevo la tabla m:

```
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Para recorrer una tabla, generalmente, usamos bucles for anidados:

```
forrowinm:
forxinrow:
        print(x, end=' ')

    print()
```

El código anterior muestra el siguiente resultado:

*1 2 3*

*4 5 6*

*7 8 9*

A veces, es útil procesar los elementos de una tabla usando sus índices:

```
x = len(m) * len(m[0])

foriiinrange(len(m)):
forjinrange(len(m[0])):
        m[i][j] = x
        print(m[i][j], end=' ')
        x -= 1

    print()
```

El código anterior modifica la tabla y muestra el siguiente resultado:

*9 8 7*

*6 5 4*

*3 2 1*

## Mutabilidad

Cuando se crea una lista de listas hay que tener en cuenta que las listas son mutables, luego los elementos (de 2º nivel) de una lista de listas (o de una tupla de listas) son mutables.

Observe cómo se crea la tabla *m* en el siguiente ejemplo:

```
m = [[0, 0, 0]] * 3
```

Tiene tres filas con tres ceros cada una. Vamos a cambiar uno de los valor es:

```
m[0][0] = -1
print(m) # [[-1, 0, 0], [-1, 0, 0], [-1, 0, 0]]
```

¿Por qué habrán cambiado los elementos *m[1][0] y m[2][0]*? Realmente la asignación:

```
m = [[0, 0, 0]] * 3
```

crea una lista con tres referencias a la misma lista [0, 0, 0], igual que el siguiente ejemplo:

```
row = [0, 0, 0]
m = [row, row, row]
```

Podemos chequear si las filas son idénticas (si referenciamos el mismo objeto, o sea, si las referencias son iguales) con el operador ***is***:

```
m = [[0, 0, 0]] * 3
print(m[0] is m[1]) # True
```

Para crear una lista de listas de forma correcta debemos primero crear una lista vacía y luego añadirle distintas filas:

```
m = []
foriinrange(3):
    m.append([0,0,0]) # en cadaiteración [0, 0, 0] construye una lista distinta
```

Ahora todo está bien:

```
m[0][0] = -1
print(m)          # [[-1, 0, 0], [0, 0, 0], [0, 0, 0]]
print(m[0] is m[1]) # False
```

## Usando funciones para crear estructuras 2D

El uso de funciones puede ser muy útil para crear estructuras 2D, como listas de listas.
La función *create_table()* del siguiente ejemplo toma las dimensiones de la tabla a crear (filas y columnas) y un valor de inicialización:

```
defcreate_table(rows, columns, value=0):
   t = []
foriinrange(rows):
      t.append([value] * columns)
returnt

my_table = create_table(2, 3)
print(my_table)                  # [[0, 0, 0], [0, 0, 0]]
my_table = create_table(4, 2, '*')
print(my_table)                  # [['*', '*'], ['*', '*'], ['*', '*'], ['*', '*']]
```

La siguiente función *create_table2()* primero crea una lista de tantos elementos como indique el parámetro *rows* inicializada con valor es *None*, por lo que aún no referencian ninguna fila. A continuación se crean las filas y se asigna su referencia a cada elemento de la lista inicial, sustituyendo al valor *None*.

```
defcreate_table2(rows, columns, value=0):
    t = [None] * rows
foriinrange(rows):
      t[i] = [value] * columns
returnt

my_table = create_table2(3, 5, 1)
print(my_table)                 # [ [1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1]]
```

## Uso de *comprehension* para crear listas de listas

Se pueden crear estructuras 2D usando comprensión:

```
t = [(i, j)fori in range(2)forj in range(3)]
print(t) # [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

Nótese, que en el ejemplo anterior, cada elemento es una tupla.

```
t = [[0] * 3foriinrange(4)]
print(t) # [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

En el ejemplo anterior, cada fila se crea independientemente (en cada iteración se re-evalúa la expresión de la izquierda).

```
t = [list(str(2 ** i))foriinrange(10)]
print(t) # [['1'], ['2'], ['4'], ['8'], ['1', '6'], ['3', '2'], ['6', '4'], ['1', '2', '8'], ['2', '5', '6'], ['5', '1',
'2']]
```

En el ejemplo anterior, las filas tienen diferentes longitud es, dado que 2 *i* va teniendo más dígitos al crecer el valor de *i*.

```
importrandom
t = [[random.randint(0, 9)] * 5foriinrange(3)]
print(t)
# [[8, 8, 8, 8, 8], [2, 2, 2, 2, 2], [7, 7, 7, 7, 7]]
```

En el ejemplo anterior todos los elementos de una misma fila tienen el mismo valor aleatorio (la lista inicial de un solo valor, aleatorio en este caso, se construye una sola vez, y luego se replica).

## Listas de listas de distintas longitudes

Hay muchas situaciones en las que necesitamos listas de listas de longitudes diferentes. La siguiente función toma como parámetros una secuencia con la longitud deseada para cada fila y un valor de inicialización:

```
defcreate_table(row_lengths, value=0):
    t = [None] * len(row_lengths)
foriinrange(len(t)):
        t[i] = [value] * row_lengths[i]
returnt
```

Primero se crea una lista conteniendo un cierto número de *None* que luego se sustituirán por las referencias a las filas que se irán creando de forma independiente, atendiendo a las longitud es indicadas por el primer parámetro.

Veamos cómo se usa:

```
t = create_table((1, 2, 3, 4, 5))
print(t)  # [[0],[0,0],[0,0,0],[0,0,0,0],[0,0,0,0,0]]

t = create_table(range(4), 'x'))
print(t) # [[], ['x'], [ 'x', 'x'], ['x', 'x', 'x']]
```

Nótese, que range(4) representa la secuencia de números 0, 1, 2, 3, por lo que la primera fila resulta una lista vacía en el segundo ejemplo.

## Tratamiento estructurado de estructuras 2D

Cuando se quiere procesar una secuencia de elementos tratando cada elemento, se sigue un esquema básico como el siguiente:

```
for elemento in lista:
```

```
    tratar elemento
```

Cuando tenemos una secuencia de secuencias (por ejemplo una lista de listas), cada elemento de dicha secuencia es, a su vez, una secuencia que, típicamente, se procesa siguiendo un mismo esquema, lo cual da lugar a un patrón de bucles anidados:

```
for lista in lista_de_listas:
    for elemento in lista:
        tratar elemento
```

Una alternativa es derivar el tratamiento de las secuencias anidadas a una función auxiliar:

```
def tratar(lista):
    for elemento in lista:
        tratar elemento
```

```
for lista in lista_de_listas:
    tratar(lista)
```

Este esquema se aplica en el siguiente ejemplo:

```
def longest_words(list_of_list_of_words):
    """Dada una lista de listas de palabras, devuelve
       una lista con las palabras más largas de cada lista
    """
    words = []
    for list_of_words in list_of_list_of_words:
        words.append(longest_word(list_of_words))
    return words

def longest_word(list_of_words):
    """Devuelve la palabra más larga de una lista de palabras"""
    current_longest = ""
    for current_word in list_of_words:
        if len(current_word) > len(current_longest):
            current_longest = current_word
    return current_longest
```

```
```

La ventaja de separar cada tratamiento en funciones diferentes, en vez de tener una única función con bucles anidados , es que acabamos teniendo un conjunto de funciones muy simples, que siguen un esquema básico, y son más fáciles de desarrollar, entender y mantener que una función compleja, con bucles anidados. Nótese, que esto se puede extender a estructuras multidimensionales (listas de listas de listas de listas de ...).

# Diccionarios

## Diccionarios

Los diccionarios, conocidos en algunos lenguajes como *arrays asociativos*, son colecciones de datos, como las secuencias, pero, a diferencia de estas, no se indexan usando un rango de números (0, 1, 2, ···) sino que usan una **clave**, que pueden ser de cualquier tipo inmutable (típicamente, strings o números). Un diccionario (tipo **dict**) es una colección **no ordenada** de parejas clave-valor.

Un diccionario inicialmente vacío se crea con unas llaves vacías:

```
d = {}
type(d)
<class 'dict'>
len(d)
0
```

También se puede crear un diccionario vacío usando sin argumentos el constructor **dict()**:

```
d = dict()
```

Un diccionario de Python se puede crear no vacío inicializándolo con uno o más elementos con la forma

*clave* : *valor*

separados por comas. El siguiente es un (mini) diccionario inglés-español, donde tanto claves como valores son strings, las primeras las palabras en inglés y los segundos su traducción al español:

```
d = {'mother': 'madre', 'father': 'padre', 'house': 'casa', 'car': 'coche'}
```

### Otras formas de inicializar un diccionario

Al constructor *dict()* se le puede pasar una secuencia de pares clave:valor. El siguiente ejemplo inicializa un (mini) diccionario inglés-eslovaco:

```
my_list = [('mother', 'matka' ), ('father', 'otec'), ('house', 'dom'), ('car', 'auto')]
d = dict(my_list)
print(d) # {'mother': 'matka', 'father': 'otec', 'house': 'dom', 'car': 'auto'}
```

También se puede usar comprehension para crear diccionarios:

```
d = {x: x**2forxin(2, 4, 6)}
print(d) # {2:4, 4:16, 6:36}
```

Cuando las claves son strings, se pueden usar como " parámetros por nombre" de *dict()*:

```
d = dict(Michael=77, Peter=66, Patrick=82)
print(d)  # {'Michael':77, 'Peter':66, 'Patrick':82}
```

## Operaciones en diccionarios

Las principales operaciones que se pueden realizar en un diccionario son:

- Almacenar un valor asociado a una clave.
- Obtener el valor asociado a una clave.
- Borrar un par clave:valor.

```
d = {'mother': 'madre', 'father': 'padre'}
d['child'] = 'hijo'
print(d) # {'mother': 'madre', 'father': 'padre', 'child': 'hijo'}
```

En el ejemplo anterior, la instrucción d['child'] = 'hijo' añade un nuevo par clave:valor al diccionario d. Si se usa una clave ya existente, el nuevo valor sustituye al almacenado previamente:

```
d['child'] = 'hija'
print(d) # {'mother': 'madre', 'father': 'padre', 'child': 'hija'}
```

Para acceder a un valor usamos su clave:

```
print(d['mother']) # madre
```

Es un error intentar acceder al diccionario usando una clave que no existe:

```
v = d['dog']
Traceback (most recent call last): File "<input>", line 7, in <module>
KeyError: 'dog'
```

Podemos prevenirlo usando el operador ***in***, que nos permite saber si una clave existe:

```
if 'dog' in d:
  print(d['dog'])
else:
  print('La clave "dog" no existe en el diccionario')
```

## Eliminar una pareja clave:valor de un diccionario

A veces, necesitamos eliminar una pareja clave:valor de un diccionario. Podemos hacerlo con el método ***pop()***.

```
d = {'mother': 'madre', 'father': 'padre', 'child': 'hijo'}
value = d.pop('mother')
print(d) # {'father': 'padre', 'child': 'hijo'}
```

El método *pop*() elimina el par clave:valor asociado a la clave pasada como parámetro y devuelve el correspondiente valor. Si no nos interesa quedarnos con el valor, podemos llamar al método *pop()* sin asignar su resultado a ninguna variable:

```
d.pop('mother')
```

La instrucción ***del*** podemos usarla también con diccionarios(obsérvese cómo ha de escribirse):

```
d = {'mother': 'madre', 'father': 'padre', 'child': 'hijo'}
del d['mother']
print(d) # {'father': 'padre', 'child': 'hijo'}
```

En ambos casos, se produce un error (*KeyError*) si la clave en cuestión no existe en el diccionario.

Para vaciar un diccionario se usa el método ***clear*()**:

```
d = {'mother': 'madre', 'father': 'padre', 'child': 'hijo'}
d.clear()
print(d) # {}
```

## El método *get()*.

A veces, es mejor usar el método *get()* para recuperar el valor asociado a una clave:

```
capitals = {'Slovakia': 'Bratislava', 'Austria': 'Vienna'}
print(capitals['Slovakia'])     # Bratislava
print(capitals.get('Slovakia')) # Bratislava
```

Supongamos que la clave no está en el diccionario:

```
capitals = {'Slovakia': 'Bratislava', 'Austria': 'Vienna'}
print(capitals.get('Hungary')) #None
print(capitals['Hungary'])
Traceback (most recent call last):
  File "my_program.py", line 2, in <module>
    print(capitals['Hungary'])
KeyError: 'Hungary'
```

El método *get()* no produce un error cuando la clave no se encuentra en el diccionario; en su lugar se devuelve el valor *None*, a menos que se especifique un valor diferente añadiendo un

segundo parámetro en la llamada:

```
print(capitals.get('Hungary', '?')) # ?
```

## Iterando en diccionarios

Aparte de obtener un valor usando una clave, podemos obtener todas las claves, todos los valores, o todos los pares *clave*:*valor* de un diccionario:

```
t = {'A': 10, 'B': 9, 'C': 9, 'D': 9, 'E': 4, 'FX': 2}
print(t.keys())
print(t.values())
print(t.items())
```

Resultado:

```
dict_keys(['A', 'B', 'C', 'D', 'E', 'FX'])
dict_values([10, 9, 9, 9, 4, 2])
dict_items([('A', 10), ('B', 9), ('C', 9), ('D', 9), ('E', 4), ('FX', 2)])
```

Los cuales son todos ellos objetos iterables:

```
forkint.keys():
   print(k, end=' ')      # A BC D E FX

forvint.values():
   print(v, end=' ')      # 10 9 9 9 4 2

fork, vint.items():     # A 10
   print(k, v)           # B 9
                        #C 9
                        # D 9
                        # E 4
                        # FX 2
```

También se puede recorrer un diccionario simplemente con:

```
forkint:
   print(k, t[k])
```

Los objetos resultantes de llamar a los métodos *keys()* ,*values()* o *items()* pueden convertirse a su vez en listas, p.e. la expresión

```
list(d.keys())
```

devuelve una lista con todas las claves del diccionario d, en un **orden** que podemos considerar **arbitrario** (recuérdese que un diccionario es una colección **no ordenada** de parejas

clave:valor). Si se necesita tener las claves ordenadas, debe usarse en su lugar la expresión

```
sorted(d.keys())
```

### Diccionarios como valores de diccionarios

Un diccionario puede ser un valor en otro diccionario:

```
s1 = {'name': 'Paul Carrot', 'address': {'Street': 'Long', 'Number': 13, 'City': 'Nitra'}}
s2 = {'name': 'Peter Pier', 'address': {'Street': 'Short', 'Number': 21, 'City': 'Nitra'}}
s3 = {'name': 'Patrick Nut', 'address': {'Street': 'Deep', 'Number': 77, 'City': 'Nitra'}}
students = [s1, s2, s3]
```

En el ejemplo anterior se crea una lista de 3 diccionarios.

El elemento students[0] es un diccionario con dos parejas clave-valor:

```
print(student[0]['name']) # Paul Carrot
print(student[0]['address']) # {'Street': 'Long', 'Number': 13, 'City': 'Nitra'}
```

El valor asociado a la clave 'address' es, a su vez, un diccionario:

```
print(student[0]['address']['Street']) # Long
print(student[0]['address']['Number']) # 13
print(student[0]['address']['City'])   # Nitra
```

# Conjuntos

## El tipo *set*

El tipo *set* de Python se usa para representar colecciones **sin orden** de elementos **sin repetición**; básicamente, implementa el concepto matemático de conjunto. Un conjunto vacío se crea usando la función constructora *set()*:

```
s =set()
```

Un conjunto no vacío se puede crear escribiendo sus elementos encerrados entre llaves, separados por coma:

```
elementos = {"agua", "aire", "fuego", "tierra"}
print(elementos)
```

```
{'agua', 'tierra', 'aire', 'fuego'}
```

*>>> Nótese que el orden de los elementos no está definido y que, por tanto, su visualización no tiene por qué coincidir con el orden en que se escribieron al crear el conjunto.*

Los elementos no pueden ser de un tipo mutable. En el siguiente ejemplo, el tercer elemento es una lista, lo que supone un error:

```
elementos = {"agua", "aire", ["fuego", "tierra"]}
```

```
Traceback (most recent call last):
  File "/home/p11049/p1.py", line 1, in <module>
    elementos = {"agua", "aire", ["fuego", "tierra"]}
TypeError: unhashable type: 'list'
```

Otra forma de crear un conjunto es pasando una colección de elementos (tupla, lista, string, diccionario, otro conjunto) a la función constructora:

```
elementos = set(["agua", "aire", "fuego", "tierra"])
print(elementos)
caracteres = set("esto es unastring")
print(caracteres)
```

```
{'agua', 'fuego', 'aire', 'tierra'}
{'a', 's', 'r', 'e', 'g', ' ', 't', 'o', 'i', 'n', 'u'}
```

*>>> Nótese que en el conjunto caracteres no hay elementos repetidos, a pesar de que sí los había en la string que se pasó para crearlo*.

En caso de que se pase un diccionario, se crea un conjunto con las claves del mismo (no las parejas clave/valor). Si se pasa un conjunto, se crea una copia del mismo.

Otra forma de copiar un conjunto es usando el método *.copy()*:

```
elementos = set(["agua", "aire", "fuego", "tierra"])
copia = elementos.copy()
```

Su cardinalidad (número de elementos de un conjunto) puede averiguarse usando la función *len()*, como en cualquier otra colección de Python:

```
elementos = set(["agua", "aire", "fuego", "tierra"])
print(len(elementos))
```

```
4
```

## Modificación de conjuntos

Los conjuntos de Python son mutables. A un conjunto existente se le pueden añadir elementos individuales usando la operación *.add()*:

```
elementos = {"agua", "aire", "fuego", "tierra"}
elementos.add("eter")
print(elementos)
```

```
{'tierra', 'eter', 'agua', 'aire', 'fuego'}
```

También se le pueden añadir elementos de una colección usando la operación *.update()*:

```
elementos = {"agua", "aire"}
elementos.update(["fuego", "tierra"])
print(elementos)
```

```
{'fuego', 'aire', 'tierra', 'agua'}
```

Por supuesto, ni *.add()* ni *.update()* añaden elementos repetidos.

Para eliminar un elemento concreto, se pueden usar las operaciones *.remove()* o *.discard()*:

```
elementos = {'fuego', 'aire', 'tierra', 'agua'}
elementos.remove("agua")
print(elementos)
```

```
{'tierra', 'fuego', 'aire'}
```

```
elementos.discard("aire")
print(elementos)
```

```
{'tierra', 'fuego'}
```

La diferencia entre *.remove()* y *.discard()* es que el primero produce un error si el elemento a eliminar no está en el conjunto, mientras que el segundo lo ignora y no hace nada:

```
elementos.remove("eter")
```

```
Traceback (most recent call last):
  File "p1.py", line 6, in <module>
    elementos.remove("eter")
KeyError: 'eter'
```

El método *.pop()* elimina un elemento al azar:

```
elementos = {'fuego', 'aire', 'tierra', 'agua'}
elementos.pop()
print(elementos)
```

```
{'agua', 'tierra', 'fuego'}
```

El método *.clear()* vacía el objeto conjunto en cuestión, eliminando todos sus elementos:

```
elementos = {'fuego', 'aire', 'tierra', 'agua'}
elementos.clear()
print(elementos)
```

```
set()
```

Obsérvese la especial visualización de un conjunto vacío. Nótese que {} representaría un diccionario vacío.

## Pertenencia e inclusión

Para comprobar si un valor pertenece a un conjunto (es uno de los elementos del conjunto) usamos el operador **in**:

```
elementos = {'fuego', 'aire', 'tierra', 'agua'}
print('aire' in elementos)
```

```
True
```

```
print('eter' in elementos)
```

```
False
```

Los operadores relacionales <, <=, >, >= permiten saber si un conjunto está incluido en otro (es un subconjunto del otro):

```
alfabeto = set("abcdefghijklmnñopqrstuvwxyz")
vocales = set("aeiou")
print(vocales < alfabeto)
```

```
True
```

Los operadores < y > requieren que sea un subconjunto estricto (se corresponden con las relaciones matemáticas $\subset$ y $\supset$), mientras que <= y >= (que se corresponden con $\subseteq$ y $\supseteq$)

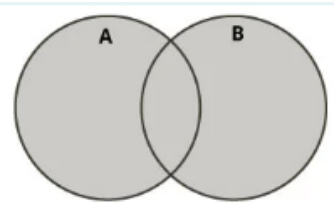
admiten también los "subconjuntos" que son iguales al conjunto que los "contiene" (subconjuntos impropios).

Entre conjuntos también se pueden usar los operadores relacionales == y !=, que determinan si dos conjuntos son iguales o distintos; dos conjuntos son iguales si tienen exactamente los mismos elementos (recuérdese que los elementos no tienen orden).

## Álgebra de conjuntos

Los conjuntos disponen de cuatro operaciones básicas: unión, intersección, diferencia y diferencia simétrica.

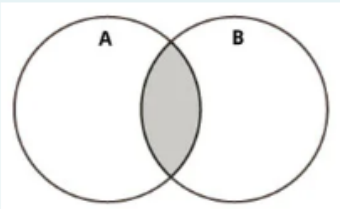
La **unión** de dos conjuntos resulta en un conjunto que tiene todos los elementos que estén en alguno(s) de los conjuntos originales, sin repetición. En Python, la unión se hace con el operador **|**(barra vertical; para escribirla, normalmente la compondremos con la tecla del 1):




```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
unión = set1 | set2
print(unión)
```

```
{1, 2, 3, 4, 5}
```

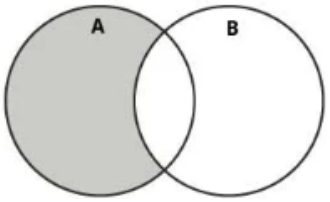
La **intersección** contiene únicamente los elementos comunes a ambos conjuntos y se hace con el operador **&**:




```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersección = set1 & set2
print(intersección)
```

```
{3}
```

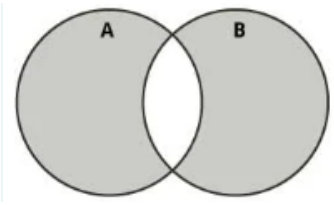
Para la **diferencia** se utiliza el operador **–** y su resultado es el conjunto formado por todos los elementos del primer conjunto que no están en el segundo:




```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
diferencia = set1 - set2
print(diferencia)
```

```
{1, 2}
```

La **diferencia simétrica** comprende todos los elementos del primer conjunto que no están en el segundo y los del segundo que no están en el primero (los elementos no comunes), y se hace con el operador **^**:




```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
diferencia = set1 ^ set2
print(diferencia)
```

```
{1, 2, 4, 5}
```

# Formateo de strings

Compare las dos capturas de pantalla que se muestran a continuación:

CAPTURA1

```
Aceite 9
Arroz 8
```

```
Azucar 10.5
Pollo 30.25
------------
Total 57.75
```

CAPTURA2

```
Aceite  9.00
Arroz   8.00
Azucar 10.50
Pollo  30.25
------------
Total  57.75
```

Ambas capturas muestran la misma información, pero la segunda muestra el texto formateado, alineando por un lado los productos y por el otro los precios, y unificando el número de dígitos decimales de estos últimos. El texto con formato resulta más legible y, en consecuencia, es más fácil interpretar la información que suministra.

## Interpolación de strings: *f-strings*

Desde la versión 3.6 de Python, el método preferente para formatear texto son los llamados "literales de string formateados" o *f-strings*, que usan como técnica la interpolación de expresiones:

```
name = "John"
age = 24
print(f'{name} tiene { age } años')  # los espacios dentro de llaves no tienen efecto
```

El código previo muestra el mensaje:

```
John tiene 24 años
```

Al anteponer una f, o una F, a un literal de tipo string, indicamos que se trata de una f-string. Una f-string es un literal string en el que se pueden intercalar expresiones de cualquier tipo, encerradas entre llaves, que se evaluarán, se convertirán a string y se insertarán en la f-string en la posición correspondiente.

```
num1 = 12
num2 = 10
print(f"sumar {num1} y {num2} da {num1 + num2}")
```

```
sumar 12 y 10 da 22
```

Si queremos incluir los caracteres de llaves en una f-string, habrá que doblarlas:

```
print(f"La palabra {{llaves}} se muestra entre llaves")
```

```
La palabra {llaves} se muestra entre llaves
```

Las f-strings, una vez evaluadas, dan como resultado una string normal, por lo que p.e. se pueden asignar a variables:

```
name = "John"
age = 24
message = f'{name} tiene {age} años'
print(message)
```

```
John tiene 24 años
```

## Modificadores de formato

La captura de pantalla que se muestra a continuación:

```
Aceite  9.00
Arroz   8.00
Azucar 10.50
Pollo  30.25
------------
Total  57.75
```

ha sido generada con el siguiente código, donde prices es un diccionario cuyas claves son nombres de productos, y los valores son sus correspondientes precios:

```
total = 0.0
for product, price in prices.items():
    print(f'{product:7}{price:5.2f}')
    total += price
print("------------")
print(f"Total  {total:5.2f}")
```

Como se puede observar, se usa una f-string en la que se interpolan las variables cuyos valores se quiere mostrar:

```
f'{product:7}{price:5.2f}'
```

pero, en lugar de estar sólo los nombres de las variables, se han añadido modificadores de formato, que son los textos que están entre los dos puntos y el cierra llaves. Los modificadores

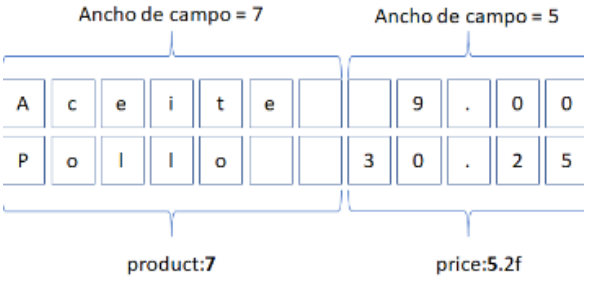
de formato controlan cómo va a aparecer el resultado de la expresión a la izquierda de los dos puntos.

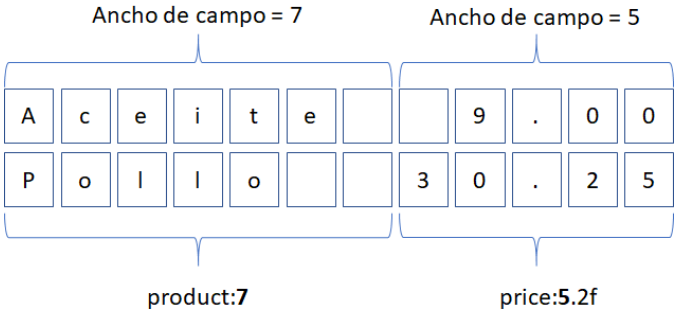
## Modificadores de formato - ancho de campo

Los modificadores de formato controlan cómo va a aparecer el resultado de la expresión a la izquierda de los dos puntos.

```
f'{product:7}{price:5.2f}'
```

El primer número del modificador de formato indica el "ancho del campo" (número de caracteres) en el que se va a mostrar el valor.




Por omisión, el texto se alinea a la izquierda del campo especificado si es una string y a la derecha si es un número, pero puede especificarse qué tipo de justificación se quiere, anteponiendo al ancho de campo los símbolos < para alinear a la izquierda, ^ para centrar y > para  alinear a la derecha:

```
value = "align left"
print(f"[{value:<20}]")
```

```
[align left          ]
```

```
value = "align center"
print(f"[{value:^20}]")
```

```
[    align center    ]
```

```
value = "align right"
print(f"[{value:>20}]")
```

```
[        align right]
```

Hay que tener en cuenta que el modificador de ancho de campo indica un mínimo, pero el campo se extenderá cuanto sea necesario para representar completamente el valor formateado. Si por el contrario fuera suficiente, por omisión el campo sobrante se rellena con espacios tal como hemos visto, si bien puede especificarse otro valor de relleno. En el caso de que el valor a representar sea un número, un cero a la izquierda del ancho de campo indica que se rellene con ceros:

```
for num in range(5, 11):
    print(f"{num:03}")
```

```
005
006
007
008
009
010
```

En cualquier otro caso, el carácter de relleno tiene que ir seguido de un código de alineación (<^>) antes del ancho de campo:

```
for num in range(5, 11):
    print(f"{num:*>3}") # > indica alineación derecha
```

```
**5
**6
**7
**8
**9
*10
```

## Modificadores de formato - Precisión

Un punto seguido de un número indica la precisión con la que se quiere formatear el valor. Véanse los siguientes ejemplos.

```
import math
print(math.pi)              # 3.141592653589793
print(1000000/3)            # 333333.3333333333
print(0.0000555)            # 5.55e-05
```

Si no se especifica formato, al convertirlos a string (en los anteriores ejemplos, automáticamente, antes de ser visualizados con print) los valores *float* se representan íntegramente, reflejando la precisión con la que se almacenan internamente (obsérvese el número de cifras para π y para el resultado de 1000000/3). La notación 5.55e-05 expresa 5.55 · 10-5.

```
print(f"{math.pi:.2}")      # 3.1
print(f"{0.00555:.2}")      # 0.0056
print(f"{555.5:.2}")        # 5.6e+02
```

La precisión 2 aquí se refiere a los 2 **primeros dígitos significativos** (se redondea apropiadamente). Se incluye al menos un dígito a la derecha del punto. No aplicable a valores de tipo *int*.

```
print(f"{math.pi:.2f}")     # 3.14
print(f"{0.00555:.2f}")     # 0.01
print(f"{555:.2f}")         # 555.00
```

La f del final indica ***f****ixed point*: la parte entera (incluso si es 0) se representará a la izquierda del punto, y a la derecha de éste se incluirán exactamente (tras el apropiado redondeo) los 2 **primeros dígitos fraccionarios** (incluso los ceros).

```
print(f"{math.pi:5.2f}")    # ␣3.14
print(f"{555:5.2f}")        # 555.00
```

Lo mismo que el caso anterior, pero en un campo de ancho 5 (al menos).

```
print(f"{str(math.pi):.2}")  # 3.
```

El valor *float* del número π se ha convertido primero a un valor de tipo string, que es lo que se formatea, en este caso recortándola y quedando únicamente los 2 **primeros caracteres** de la misma (nótese la diferencia con respecto al caso en que la misma especificación de formato se aplica sobre un valor de tipo numérico).

## Modificadores de formato - Notación numérica

Al final del modificador de formato se puede añadir una letra para especificar cómo queremos interpretar el tipo de datos. Véanse los siguientes ejemplos:

```
print(f"{12:5o}")
```

```
  14
```

El valor entero 12 (por omisión, en la usual base diez) se imprime en octal (base ocho, que solo usa los dígitos del 0 al 7), en un campo de tamaño 5.

```
print(f"{12:5x}")
```

```
   c
```

```
print(f"{12:5X}")
```

```
C
```

El valor entero se imprime en hexadecimal (base dieciséis, que usa los dígitos 0-9 y a-f). Las letras 'a', 'b', 'c', 'd', 'e' y 'f' de la representación hexadecimal se muestran en mayúscula si el modificador 'X' está en mayúscula.

```
print(f"{12:5.1f}")
```

```
 12.0
```

Un valor, entero o flotante, se imprime como *float* (si el valor es entero, se añaden ceros como parte fraccionaria)

```
print(f"{-140:5.1e}")
```

```
-1.4e+02
```

```
print(f"{0.014:5.1E}")
```

```
1.4E-02
```

Un valor, entero o flotante, se imprime en notación científica (coma flotante) indicando el exponente de la potencia de diez: −140 = −1.4×102, 0.014 = 1.4×10−2. Obsérvese que se muestra "e" o "E" según el modificador.

```
print(f"{12:5d}")
```

```
   12
```

Un valor *entero* se imprime en formato decimal (la usual base diez, que usa los dígitos del 0 a 9).

### Carácter de signo

Podemos controlar qué ocurre al mostrar el signo cuando el número no es negativo. Ejemplos:

```
print(f"{-12:5d}")
```

```
  -12
```

Por omisión, el carácter para el signo (– o +) sólo se muestra si el número es negativo.

```
print(f"{-12:+5d}")
```

```
  -12
```

```
print(f"{12:+5d}")
```

```
  +12
```

Si al principio de los modificadores de formato añadimos el carácter '+' se mostrará, tanto el signo '–' como el '+', en función de si el valor numérico es negativo o  no.

```
print(f"{-12:-2d}")
```

```
-12
```

```
print(f"{12:-2d}")
```

```
12
```

```
print(f"{-12: 2d}")
```

```
-12
```

```
print(f"{12: 2d}")
```

```
␣12
```

Añadir un carácter '−' no afecta al resultado (sólo se muestra el signo cuando el valor es negativo), pero añadir un espacio en su lugar muestra un espacio (que aquí hemos representado por␣) en lugar del signo cuando el valor no es negativo.

### El método format()

Una alternativa a la interpolación de strings que, aunque propia de versiones de Python anteriores a la introducción de este mecanismo, sigue disponible es el método format():

```
num1 = 12 / 7
print("Primero = {:4.2f}, segundo = {:2X}".format(num1, 10))
```

```
Primero = 1.71, segundo =  A
```

Como se ve en el ejemplo, es muy similar a la interpolación de strings, salvo que las expresiones cuyos valores queremos incluir en la string no se colocan en su sitio dentro de ella, sino que se pasan como parámetros del método *format()*, ocupando el lugar que tienen reservado en la string, en principio por orden de aparición.

Existe la posibilidad de dar un nombre a cada campo, lo que independiza su orden en la string del orden en que se pasan al método format().

```
print("Primero = {num1:4.2f}, segundo = {num2:2X}".format(num1=num1, num2=10))
print("Segundo = {num2:2X}, primero = {num1:4.2f}".format(num1=num1, num2=10))
```

# Más sobre strings

## Strings

Las strings representan secuencias contiguas de caracteres. Los valores literales de tipo string (*str*), se encierran entre comillas dobles o simples:

```
var1 = "hola"
var2 = 'hola'
```

```
str_var = "Hello world"
print(str_var)        # Muestra "Hello world" (sin las comillas delimitadoras)
```

Se puede crear una string multilínea usando triples comillas (simples o dobles):

```
var3 = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
```

(los caracteres de salto de línea serán parte de la string).

La longitud de una string (número de caracteres) se conoce usando la función len():

```
str_var = "Hello world"
print(len(str_var))     # Muestra elvalor entero 11
```

## Secuencias de escape

Se pueden introducir caracteres especiales en las strings (tabuladores, saltos de línea, caracteres por código, ...) usando secuencias de escape, que son secuencias de caracteres que empiezan con el carácter '\' (llamado barra invertida o *backslash*).

**\' - Comilla simple** (evita confusión con comilla simple delimitadora)

**\" - Comilla doble** (evita confusión con comilla doble delimitadora)

**\\ - backslash** (evita confusión con el propio backslash de secuencia de escape)

**\a - Sonido de campana**

**\b - Backspace (retroceso)**

**\v - Vertical tab**

**\xhh - Carácter por valor hexadecimal**

**\n - Newline (nueva línea)**

**\N{name} - Carácter Unicode por nombre**

**\ooo - Carácter por valor octal**

**\r - Carriage return** (al mostrar la string, sobrescribe desde el principio de la línea actual)

**\uxxxx - Carácter Unicode por valor hexadecimal (16 bits)**

**\Uxxxxxxxx - Carácter Unicode por valor hexadecimal (32 bits)**

## Mayúsculas y minúsculas

Existen métodos para saber si las letras que contiene una string son mayúsculas o minúsculas y para hacer cambios de un caso al otro.

El método **isupper()** devuelve True si las letras de una string son mayúsculas; el método **islower()** hace lo propio si son minúsculas.

```
test_string = "en un lugar de La Mancha de cuyo nombre"
a = test_string.isupper()    # False
```

El método **upper()** devuelve una nueva string, como la original, pero con sus letras transformadas a mayúsculas; lo contrario hacen los métodos **lower()** y **casefold()** (ambos transforman a minúsculas).

```
b = test_string.upper()      # "EN UN LUGAR DE LA MANCHA DE CUYO NOMBRE"
```

El método **capitalize()** devuelve una nueva string, como la original, pero poniendo en mayúscula la letra inicial y en minúscula el resto.

```
c = test_string.capitalize() # "En un lugar de la mancha de cuyo nombre"
```

El método **title()** devuelve una nueva string igual que la original pero poniendo en mayúscula la letra inicial de cada palabra y en minúscula el resto. Existe un método llamado **istitle()** para verificar si una string sigue este formato.

```
d = test_string.title()      # "En Un Lugar De La Mancha De Cuyo Nombre"
```

El método **swapcase()** devuelve una nueva string invirtiendo mayúsculas y minúsculas.

```
f = d.swapcase()             # "eN uN lUGAR dE lA mANCHA dE cUYO nOMBRE"
```

## Categorías de caracteres

Existen una serie de métodos para conocer qué tipo de caracteres forman una string.

#### isalnum()

Devuelve True si todos los caracteres de la string son alfanuméricos.

#### isalpha()

Devuelve True si todos los caracteres de la string están en el alfabeto.

#### isdecimal()

Devuelve True si todos los caracteres de la string son decimales.

#### isdigit()

Devuelve True si todos los caracteres de la string son dígitos.

**isidentifier()**

Devuelve Tue si la string es un identificador. Una string se considera un identificador válido si sólo contiene letras alfanuméricas (a-z, A-Z y 0-9), o guiones bajos (_). Un identificador válido no puede comenzar con un número o contener espacios.

**isnumeric()**

Devuelve True si todos los caracteres de la string son numéricos

**isprintable()**

Devuelve True si todos los caracteres de la string son imprimibles.

**isspace()**

Devuelve True si todos los caracteres de la string son espaciadores (espacios, tabuladores, saltos de línea, ...)

## Acceso a los componentes de una string

Se puede acceder a un carácter concreto de una string escribiendo la string, o el nombre de la variable que la referencia, seguida del índice de la posición que ocupa el carácter deseado entre corchetes. El índice de la primera posición es 0.

```
str_var = "Hello world"
print(str_var[0])      # Muestra (sin comillas) el primer carácter de str_var:'H'
print(str_var[4])      # Muestra (sin comillas) el quinto carácter de str_var:'o'
```

También se pueden usar índices relativos a la longitud de la string:

```
str_var = "Hello world"
print(str_var[-1])     # Muestra el último carácter de str_var:'d'
print(str_var[-5])     # Muestra el quinto carácter, empezando por el final, de str_var:'w'
```

Se puede obtener una substring (subsecuencia) de una string usando el operador de segmento, o slice, ([:]). El segmento deseado se indica mediante el índice del primer elemento y el índice de la posición siguiente al último elemento. Si se omite el segundo valor, se toma el segmento desde el primer índice hasta el final; si se omite el primero, toma desde el principio hasta la posición del índice anterior al indicado:

```
str_var = "Hello world"
print(str_var[2:5])    # Muestra los caracteres de 3º al 5º: "llo"
print(str_var[2:])     # Muestra los caracteres a partir del 3º: "llo world"
print(str_var[:5])     # Muestra los caracteres hasta la 5ª posición: "Hello"
```

Usando un paso de -1 se puede obtener la string invertida:

```
str_var = "Hello world"
print(str_var[::-1])    # Muestra "dlrow olleH"
```

## Concatenación de strings

El diccionario de la RAE define concatenar como "unir dos o más cosas"; en el caso de las strings, concatenar dos strings consiste en formar una nueva string juntando, en orden, dos preexistentes.

En Python existen dos operadores de concatenación: el signo más (**+**) es el operador de concatenación de strings y el asterisco (*****) es el operador de repetición (equivalente a realizar una concatenación repetidas veces).

```
str_var = "Hello world"
print('a' + 'a')       # Muestra "aa"
print(str_var + "TEST") # Muestra elvalor de str_var concatenado con "TEST": "Hello world TEST"
print('ab' * 5)        # Muestra "abababababab"
print(str_var * 2)     # Muestra "Hello worldHello world"
```

Podríamos decir que la repetición de strings es a la concatenación de strings lo que la multiplicación de números entero es a la suma de números enteros.

Nótese que la concatenación junta directamente las strings, sin añadir ningún carácter en medio.

## Localización de una substring

El método **find()** permite localizar una substring dentro de una string:

```
s1 = "Esto es unastring de prueba"
print(s1.find("una"))  # 8
```

El método find() devuelve el índice del primer carácter de la primera aparición de la string pasada como parámetro ("una" en el ejemplo) en la string sobre la que se aplica (la referenciada por s1 en el ejemplo).

```
s1 = "Esto es unastring, es una secuencia, pero no una lista"
p = s1.find("una") # 8
```

En caso de que la string a buscar no se encuentre en la string donde se busca, el método find() devuelve el valor -1.

También se puede usar el método **index()**, aplicable a cualquier tipo de secuencia (strings, listas, tuplas, ...).

```
s1 = "Esto es unastring de prueba"
print(s1.index("una"))  # 8
```

La diferencia es que, si la string buscada no se encuentra, el método index() produce un error. Una forma de evitarlo es usar el operador in para comprobar primero que está la string buscada:

```
s1 = "Esto es unastring de prueba"
if "una" in s1
   print(s1.index("una"))
```

Si sólo nos interesa saber si la string buscada está o no en la string donde se busca, pero no necesitamos conocer su índice, basta con el operador in:

```
print("una" in s1)  # True
```

El método **find()** admite dos parámetros adicionales opcionales:

- start: índice de la posición donde comenzar la búsqueda (el valor por omisión es 0)
- end: índice de la posición donde dejar de buscar (el valor por omisión es la longitud de la string donde se busca)

Por ejemplo:

```
print(s1.find("e ", 10, 24))  # 20
```

## Otros métodos de localización

Los métodos **rfind()** y **rindex()** localizan la última ocurrencia de la string buscada, a diferencia de sus contrarios, find() e index(), que localizan la primera.

El método **startswith()** devuelve True si la string pasada como parámetro coincide con el comienzo de la string sobre la que se busca. Lo equivalente hace **endswith()** por el final.

El método **count()** devuelve el número de apariciones de la string buscada en la string de búsqueda.

El método **partition()** devuelve una tupla con las tres partes en que podemos considerar dividida la string en que busca: la substring anterior, la primera ocurrencia de la substring encontrada y la substring posterior; si no se encuentra, las tres partes serán la propia string buscada y dos strings vacías. El método **rpartition()** hace lo mismo desde el final, con la última ocurrencia de la string buscada.

## El método join()

El método join() devuelve una string concatenando las sucesivas strings contenidas en la secuencia pasada como parámetro, usando como separador la string con la que se llama al

método.

Ejemplo:

```
words = ['sun', 'is', 'shining']
espacio = ' ' # un espacio
message = espacio.join(words)
```

```
print(message)              # "sun is shining" (sin comillas)

importrandomnumbers = [str(random.randint(1, 9)) for i in range(5)]
print(numbers)              # p.e. ['9', '1', '9', '3','2']
excercise = ' + '.join(numbers)
print(excercise, '= ')      # "9 + 1 + 9 + 3 + 2 = " (sin comillas)
```

En los ejemplos anteriores, las secuencias pasadas al método join() son listas, pero pueden ser cualquier clase de secuencia, por ejemplo, otra string, en cuyo caso, el separador se intercalaría entre los caracteres de la misma:

```
s1 = "ABC"
print("-".join(s1)) # "A-B-C" (sin comillas)
```

## El método split()

El método split() crea una lista cuyos elementos son sucesivos trozos de la string sobre la que se aplica.

Si se le llama sin parámetros, los elemento de la lista son los trozos que en la string original están **delimitados** por espaciadores (*whitespaces*), esto es, cualesquiera separadores estándar de texto, tales como espacios en blanco o/y tabuladores:

```
a = 'esto es un ejemplo'
b = a.split()  # ['esto', 'es', 'un', 'ejemplo']
```

Se puede especificar un **separador** entre los trozos (se consideraría que existe un trozo string vacía entre dos separadores seguidos). En el siguiente ejemplo es la coma:

```
a = 'esto es, un ejemplo, de split, conparámetro'
b = a.split(',')  # ['esto es', ' un ejemplo', ' de split', ' conparámetro']
```

También se puede especificar, con un parámetro adicional, el número máximo de "cortes" a realizar. El siguiente ejemplo devuelve una lista con tres elementos, dado que se pasa un 2 como segundo parámetro:

```
a = 'esto es, un ejemplo, de split, conparámetros'
b = a.split(',', 2)  # ['esto es', ' un ejemplo', ' de split, conparámetros']
```

## El método replace()

El método replace(), devuelve una string que es igual a la string sobre la que se aplica, cambiando las ocurrencias de una substring por otro valor.

Supongamos la siguiente asignación:

```
a = "Susanita tiene un ratón"
```

Supongamos ahora que el ratón de Susanita se escapó:

```
b = a.replace("tiene", "tenía") # b = "Susanita tenía un ratón"
```

El método replace() sustituye cada una de las apariciones:

```
a = "Un globo, dos globos, tres globos"
b = a.replace("globo", "elefante") # b = "un elefante, dos elefantes, tres elefantes"
```

## Recorte y relleno: los métodos strip(), lstrip(), rstrip(), ljust(), rjust(), center() y zfill()

A veces tenemos una string con espacios al principio o al final que no nos resultan útiles y queremos eliminar. El método **strip()** elimina los espaciadores de los extremos de una string:

```
a = "string con espacios   "
b = a.strip() # "string con espacios"
```

Los métodos **lstrip()** y **rstrip()** hacen lo mismo, pero sólo en uno de los extremos (izquierdo - left o derecho - right):

```
left = a.lstrip()   # "string con espacios   "
right = a.rstrip()  # "string con espacios"
```

También existen métodos para justificar una string en un tamaño determinado, ajustándola a la izquierda o a la derecha y rellenando el espacio sobrante con un carácter especificado:

```
a = "hola"
b = a.ljust(10, "_") # "hola______"
c = a.rjust(10, "_") # "______hola"
```

Combinando los métodos ljust() y rjust(), se puede lograr una justificación "centrada", aunque existe un método, center(), para hacer eso:

```
mid = 5 + len(a) // 2
d = a.ljust(mid, "_").rjust(10, "_") # "___hola___"
```

```
d1 = a.center(10, "_")              # "___hola___"
```

Por su parte, el método **zfill()** rellena una string añadiéndole ceros por la izquierda hasta alcanzar un tamaño determinado:

```
num = "12"
num1 = num.zfill(5) # "00012"
```

Obviamente, siempre es posible rellenar con cualquier carácter usando operaciones más básicas:

```
num = "12"
num1 = 'X' * (5 - len(num)) + num # "XXX12"
```

# Expresiones regulares

Numerosos problemas de tratamiento de ristras requieren la localización en un texto de subristras que cumplen un determinado patrón: fechas, números de DNI, números de la seguridad social, e-mails, direcciones web, etc. La búsqueda exacta que proporcionan operaciones como el método *find()* de las strings resulta insuficiente para resolver este tipo de problemas.

Una expresión regular es una secuencia de caracteres que define un patrón de búsqueda en una string. Por ejemplo, la secuencia:

```
^r..o$
```

define un patrón para una string de cuatro caracteres, empezando por la letra *r* (^r), terminando por la letra *o* (o$) y con dos caracteres en medio (..). Strings que encajan en este patrón de búsqueda son, por ejemplo: 'ramo', 'rito', 'rato', 'ralo', ...

Python tiene un módulo llamado **re** que proporciona herramientas para trabajar con expresiones regulares:

```
import re
pattern = '^r..o$'
test_string = 'ramo'
result = re.match(pattern, test_string)
```

```
if result:
  print("Búsqueda exitosa")
else:
  print("Búsqueda fallida")
```

Nótese que las expresiones regulares, al ser secuencias de caracteres, se representan en un programa usando valores del tipo string (la variable *pattern* en el ejemplo).

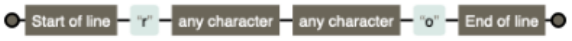
La función *match()* del módulo *re* devuelve un objeto de tipo *match* (del que hablaremos más adelante) si encuentra una coincidencia, o *None* si no la encuentra.

## Metacaracteres

Las expresiones regulares son una combinación de caracteres normales y metacaracteres. Estos últimos tienen un significado especial, distinto de su valor facial. En la expresión:

```
^r..o$
```

los caracteres '^', '.' y '$' son metacaracteres; '^' significa "principio de línea", '.' significa "cualquier caracter" y '$' significa "final de línea".




Si necesitamos que un metacaracter se busque por su valor como carácter, tenemos que "escaparlo" con el carácter '\':

```
^r..o\.$
```

La expresión significa "una string que ocupa una línea, empieza por la letra 'r', seguida de dos caracteres cualesquiera y termina con la secuencia 'o.'" ("ramo.", "reto.", "ralo.", ...):


## Metacaracteres - clases de caracteres

El carácter '.' es un metacarácter que se usa como comodín para indicar que el patrón admite cualquier carácter (salvo fin de línea) en esa posición.

Varias secuencias de caracteres concretas, que empiezan con el carácter '\' se usan para indicar distintas clases de caracteres.

La secuencia '\d' significa "cualquier dígito decimal":

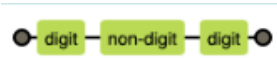
\d\d-\d\d-\d\d\d\d


"19-12-1978", "22-12-2008", "41-20-9999", ...

---

La secuencia '\D' significa "cualquier carácter que no sea un dígito decimal":

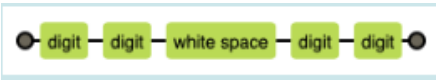
\d\D\d



"1A2", "3-5", "1 5",...

La secuencia '\s' significa "un carácter espaciador", que puede ser p.e. un '\t' - tabulador, '\n' - salto de línea, '\r' - carriage return, o un ' ' - espacio.

\d\d\s\d\d




"12 32", "12\t21", "33\n35",...

La secuencia '\S' significa "cualquier carácter que no sea un espaciador".

La secuencia '\w' significa "un carácter de palabra" (la 'w' es de *word*). Se consideran "caracteres de palabra" las letras mayúsculas y minúsculas, los dígitos decimales, del '0' al '9', y el guión bajo o underscore '_'.

\w\w\w


"abc", "123", "A_1",...

La secuencia '\W' significa "cualquier carácter que no sea de palabra".

## Metacaracteres - cuantificadores

Algunos metacaracteres sirven como cuantificadores, indicando repeticiones en el patrón.

El carácter asterisco '*' indica cero o más ocurrencias del patrón que le precede:

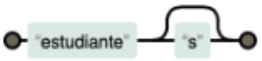
```
muu*cho
"mucho", "muucho", "muuuucho",...
```


El carácter '+' indica una o más ocurrencias del patrón que le precede:

```
mu+cho
"mucho", "muucho", "muuuucho",...
```

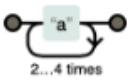
El carácter '?' indica cero o una ocurrencia del patrón que le precede:

```
estudiantes?
"estudiante", "estudiantes"
```




Unas llaves con dos números en medio separados por comas, '{m,n}' se usan para indicar un rango de ocurrencias del patrón que las precede; significa "como mínimo m ocurrencias y como máximo n"):

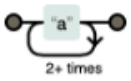
```
a{3,5}
"aaa", "aaaa", "aaaaa"
```




*Atención a cómo se muestra en el diagrama, que especifica "2...4 times". Con ello se pretende indicar que ese camino de "vuelta atrás" se ha de recorrer, como mínimo, 2 veces y, como máximo, 4 veces, por lo que el número de veces que hemos pasar por la "a" cuando recorremos el diagrama desde su extremo izquierdo al derecho será entre 3 y 5 veces, que son los números que indicamos en la expresión regular. Por tanto, las strings mostradas son las únicas que encajan con ese patrón, ya que cualquier otra string no permitiría hacer un recorrido como se ha descrito.*

Unas llaves con un número en medio seguido de una coma, '{m,}' indica un número mínimo de apariciones del patrón precedente:

```
a{3,}
"aaa", "aaaa", "aaaaa", "aaaaaaaaaa",...
```




Unas llaves con un número en medio, '{m}' se usa para indicar un número exacto de apariciones:

```
\d{2}-\d{2}-\d{4}
```

```
"12-10-1978", "15-03-2020", "43-29-0098", ...
```

*Atención de nuevo, por ejemplo "once": se ha de recorrer el camino de vuelta atrás una vez (y no más) por lo que deberán haber exactamente dos dígitos seguidos.*

## Greedy vs nongreedy

Los cuantificadores repetitivos, como '+' o '*', son "avariciosos" (greedy), esto quiere decir que intentan tomar el máximo número de repeticiones posible, lo cual puede ser problemático en ciertas circunstancias. En el siguiente ejemplo, se intenta obtener, separadas, todas las etiquetas de un documento html; una etiqueta del lenguaje html empieza con el carácter '<', seguido de cualquier número de caracteres y finaliza con el carácter '>':

```
import re
pattern = '<.*>'
test_string = '<html><head><title>Título</title></head><body>Cuerpo del documento</b
ody></html>'
print(re.findall(pattern, test_string))
```

Aunque el patrón usado parece correcto (por ejemplo, '<html>' y '<head>' encajan aisladamente), el carácter "avaricioso" del cuantificador '*' hace que todo el texto (hasta el último '>') se considere como una única "etiqueta" html:

```
['<html><head><title>Título</title></head><body>Cuerpo del documento</body></html
>']
```

Para evitarlo, en Python se puede incluir una interrogación detrás del '*', para indicarle que se extienda lo menos posible, en vez de lo más posible:

```
import re
pattern = '<.*?>'
test_string = '<html><head><title>Título</title></head><body>Cuerpo</body></html>'
print(re.findall(pattern, test_string))
```

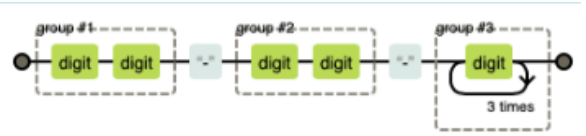
Ahora se obtiene:

```
['<html>', '<head>', '<title>', '</title>', '</head>', '<body>', '</body>', '</html>']
```

## Metacaracteres - grupos

Se pueden agrupar porciones de una expresión regular usando paréntesis:

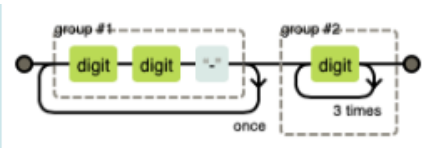
```
(\d\d)-(\d\d)-(\d{4})
```




El agrupamiento tiene varias ventajas. La primera es que permite identificar subexpresiones (grupos) a las que se puede acceder por separado en las operaciones de tratamiento de strings que se usen. Suponiendo que el ejemplo anterior se usara para localizar posibles fechas *día-mes-año* en un texto, al usar paréntesis el día queda identificado como grupo 1, el mes como grupo 2 y el año como grupo 3; la fecha completa sería el grupo 0.

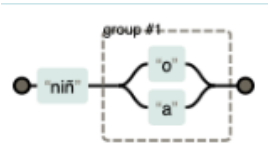
El agrupamiento también permite compactar las expresiones. Las ocurrencias encontradas por el patrón del ejemplo anterior también pueden ser encontradas por el siguiente patrón (dicho de otra manera, ambos patrones el del ejemplo anterior y el que se muestra a continuación son equivalentes):

```
(\d\d-){2}(\d{4})
```

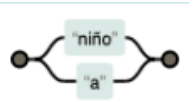



El agrupamiento también permite delimitar un grupo de alternativas. Se usa la barra vertical '|' para separar dos (o más) trozos del patrón indicando así que son alternativos entre sí; de hecho, si no existiese la posibilidad de agrupar, sería difícil o engorroso expresar determinados patrones de alternativas. Obsérvense los dos siguientes patrones:

```
niñ(o|a)
```




```
niño|a
```




Al no usarse agrupamiento en este último caso, todo lo que está a la izquierda de la barra es alternativa de lo que está a la derecha. Esto es debido a que el "operador invisible" de concatenación, en este caso "operando" sobre los caracteres que conforman el subpatrón 'niño' tiene precedencia sobre el operador, este sí expreso, '|', de hecho este operador tiene menor **precedencia** también que '*' y '+', los cuales a su vez tienen mayor precedencia que

la concatenación; así pues, como en las expresiones aritméticas, los paréntesis nos sirven para forzar una interpretación distinta de la determinada por las precedencias.

Para encontrar la misma string que con el primer patrón, sin usar agrupamiento, habría que usar un patrón más largo:

```
niño|niña
```


Podemos entender este otro patrón como similar al resultado de "quitar paréntesis" en el caso de una expresión aritmética donde en lugar del '|' tuviéramos un operador + y donde la multiplicación, que correspondería con la concatenación, es implícita, sin expresar el operador, como muchas veces hacemos en matemáticas: p.e. *abc(x+y) = abcx+abcy*.

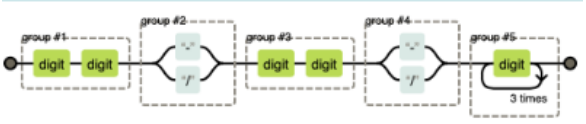
## Metacaracteres - backreferences

Otra ventaja de los agrupamientos, es que permiten hacer "backreferences". Supongamos que queremos usar el patrón del primer ejemplo para buscar posibles fechas en un texto:

```
(\d\d)-(\d\d)-(\d{4})
```


Este patrón permite encontrar fechas que usen guiones como separadores ("30-06-1979"), pero no fechas que usen barras ("30/06/1970"). Podemos usar alternativas para contemplar ambos separadores:

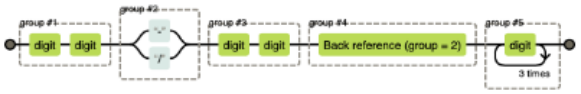
```
(\d\d)(-|/)(\d\d)(-|/)(\d{4})
```




Este patrón acepta como separadores guiones o barras, pero permite mezclarlos en una misma fecha ("30-06/1970", "30/06-1970"), lo que normalmente no consideraríamos correcto.

Una "backreference" consiste en insertar en una expresión regular una referencia a un grupo anterior, usando su número precedido de una barra invertida '\' (*backslash*):

```
(\d\d)(-|/)(\d\d)(\2)(\d{4})
```




En este caso, la especificación para el separador de la derecha hace referencia con \2 a que debe ser igual a la sub-string concreta que encajó con el grupo 2, esto es, si el separador de la izquierda es un guión, el otro deberá serlo también, y si es una barra, el otro deberá ser también una barra.

```
import re
pattern = '(\d\d)(-|/)(\d\d)(\\2)(\d{4})'
test_string = '30/06/1970'
result = re.match(pattern, test_string)
```

```
if result:
  print("Búsqueda exitosa")
else:
  print("Búsqueda fallida")
```

Obsérvese que hemos tenido que duplicar la barra antres del 2, de lo contrario, con una sola barra, '\2' ¡sería tomado como un carácter expresado mediante una secuencia de escape!

## La barra para metacaracteres de expresión regular vs secuencias de escape de string

En el siguiente trozo de código:

```
import re
pattern = '(\d\d)(-|/)(\d\d)(\\2)(\d{4})'
test_string = '30/06/1970'
result = re.match(pattern, test_string)
```

```
if result:
  print("Búsqueda exitosa")
else:
  print("Búsqueda fallida")
```

Podemos observar que, aunque la expresión regular que representa lo que queremos buscar es:

```
'(\d\d)(-|/)(\d\d)(\2)(\d{4})'
```

A la variable *pattern*, que representa en el código el patrón de búsqueda, le hemos asignado:

```
'(\d\d)(-|/)(\d\d)(\\2)(\d{4})'
```

Nótese, que hemos **duplicado** la barra que está antes del dos en la backreference.

La razón es que, mientras que en una expresión regular la barra se usa para representar ciertos metacaracteres, la misma barra se usa en los literales normales de string para expresar ciertas secuencias de escape. Así, en una expresión regular '\2' es una referencia al grupo 2 previo pero '\n' es una secuencia de dos caracteres (barra seguido de 'n'). Pero al usar un literal normal de string para emplearlo como expresión regular tenemos el problema de que '\2' será tomado por una secuencia de escape que representa el caracter cuyo código en octal es 2 y '\n' como el carácter de nueva línea (*newline*). Para solucionarlo, en estos casos concretos de posible interpretación como secuencia de escape deberemos duplicar la barra, ya que en un literal de string '\\' se toma como una sola barra. En las secuencias de metacaracteres que no coincidan con secuencias de escape no hace falta duplicar la barra.

Una alternativa es indicar que el literal de string es una ***raw string***, en la que los caracteres se toman tal cual (aunque por tanto si lo necesitásemos no podríamos expresar caracteres especiales mediante secuencias de escape) anteponiendo una letra **r** a las comillas (simples o dobles):

```
import re
pattern = r'(\d\d)(-|/)(\d\d)(\2)(\d{4})'
test_string = '30/06/1970'
result = re.match(pattern, test_string)
```

```
if result:
  print("Búsqueda exitosa")
else:
  print("Búsqueda fallida")
```

Por lo demás, la string resultante es del mismo tipo de datos *str* que el de un literal normal, y podemos concatenar combinando ambas modalidades, a conveniencia.

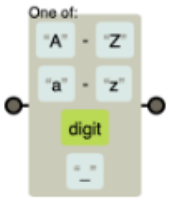
## Metacaracteres - conjuntos y rangos de caracteres

Los corchetes se pueden usar en una expresión regular para expresar "cualquier carácter de un conjunto". Por ejemplo, el siguiente patrón permite que el último carácter sea una 'a' o una 'o':

```
niñ[oa]
```

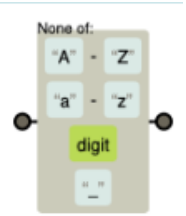
Se puede usar el guión '-' entre dos caracteres dentro de los corchetes para indicar un rango. El siguiente patrón admite una letra mayúscula (rango A-Z), una letra minúscula (rango a-z), un dígito, o un guión bajo:

```
[A-Za-z\d_]
```




Un carácter '^' (acento circunflejo) como primer carácter entre los corchetes sirve para negar su contenido (conjunto complementario), es decir indica "cualquier carácter que no esté en el conjunto indicado a continuación". Nótese la expresión "None of" en la parte superior de la imagen adjunta al siguiente patrón:

```
[^A-Za-z\d_]
```




## Metacaracteres - delimitadores

Algunos metacaracteres sirven como delimitadores, esto es, no representan carácter alguno por sí msimos, sino que expresan requisito que debe cumplir la string en ese punto del patrón.

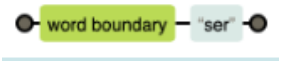
El acento circunflejo '^', cuando no está dentro de corchetes, indica que ese punto del patrón debe corresponder con principio de línea. Dualmente, el signo de dólar '$' indica fin de línea. El siguiente patrón abarca una línea completa, de principio a fin:


```
^r..o$
```

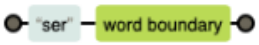
También se puede usar la secuencia '\A' que indica principio de string. Nótese que esto es distinto de principio de línea sólo en el caso de strings multilínea.

La secuencia '\b' indica extremo de "palabra" (ésta en el sentido de '\w' visto anteriormente). En los siguientes ejemplos se muestran en azul el tramo de la string en cuestión que encaja con el patrón de la izquierda conteniendo '\b' o su opuesto '\B':

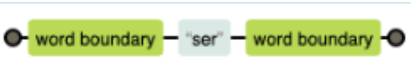
\bserp.e. en "alservir"




ser\bp.e. en "al coser"



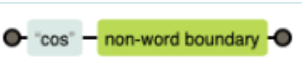

\bser\bp.e. en "elser o la nada"



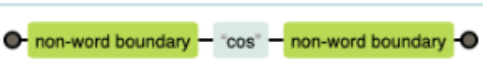

La secuencia '\B', por el contrario, indica que ese punto no debe ser extremo de palabra:

\Bcosp.e. en "descoser", "cascos"


cos\Bp.e. en "descoser", "coser"




\Bcos\Bp.e. en "descoser"




# El módulo re

Python tiene un módulo llamado **re** para trabajar con expresiones regulares. Este módulo ofrece diferentes funciones para buscar un patrón expresado por una expresión regular en una string, e incluso para sustituir por otras substrings las apariciones que encajen con un patrón determinado.

Muchas de estas funciones devuelven un objeto de coincidencia que contiene diversa información sobre la ocurrencia encontrada. Por ejemplo, la función match() busca

la ocurrencia de un patrón al principio de una string y devuelve un *objeto match*, si la hay, o None, si no la hay:

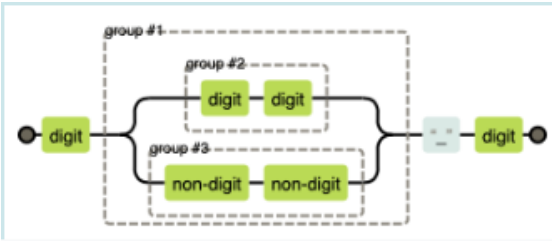
```
import re
pattern = '(\d\d)(-|/)(\d\d)(\\2)(\d{4})'
test_string = '30/06/1970 es una fecha muy popular'
result =re.match(pattern, test_string)
if result:
    print(result)         # <_sre.SRE_Match object; span=(0, 10), match='30/06/1970'>
    print(result.start())  # 0
    print(result.end())    # 10
    print(result.span())   # (0, 10)
    print(result.group(0)) # 30/06/1970
    print(result.group(5)) # 1970
```

Como vemos, los objetos de coincidencia ofrecen diferentes métodos para extraer la información de un objeto de ese tipo:

| | |
|---|---|
| **start()** | Índice del comienzo de la ocurrencia encontrada, que para el método match() es siempre 0 |
| **end()** | Índice de finalización de la ocurrencia, la cual va, como es usual, desde start() hasta end()-1, ambos inclusive |
| **span()** | Una tupla con los valores de start() y end() |
| **group()** | La substring que ha encajado con el grupo cuyo número se pasa por parámetro, además, group(0) devuelve la ocurrencia completa |

Hay que tener cuidado con el uso del método group(); el grupo 0 siempre existe pero, dependiendo de la expresión, otros grupos podrían no existir (se retorna None), por lo que habría que preguntar si existen, antes de intentar acceder a ellos. En el siguiente ejemplo, los grupos 2 y 3 son excluyentes, de manera que si en una ocurrencia aparece uno de ellos, no puede aparecer el otro. Por ejemplo, dado el siguiente patrón:

```
\d((\d\d)|(\D\D))_\d
```




El grupo 2 está en la string "123_4", donde no está el grupo 3, y el grupo 3 está en la string "1AB_4", donde no está el grupo 2.

Por el contrario, varias substrings pueden encajar con un mismo grupo si éste se encuentra dentro de una estructura repetitiva (p.e. con *): group() nos devolverá **solo la última** de tales substrings.

## La función search()

Si la función match() devuelve un objeto match para la ocurrencia del patrón al principio de la string de búsqueda, la función search() devuelve un objeto match para la primera ocurrencia del patrón en la string de búsqueda, independientemente de la posición donde se encuentre. Si no hay ninguna ocurrencia del patrón en la string de búsqueda, devuelve None, como la función match():

```
import re
pattern = '\d\d(-|/)\d\d\\1\d{4}'
test_string = 'palabra1 30/06/1970 PALABRA2 12-12-2008 palabra3'
result =re.search(pattern, test_string)
print(result)
```

El ejemplo anterior muestra:

```
<_sre.SRE_Match object; span=(9, 19), match='30/06/1970'>
```

## La función finditer()

La función finditer() devuelve una secuencia iterable de objetos *match* con todas las ocurrencias (no solapadas) del patrón en la string donde se busca:

```
import re
```

```
pattern = '\d\d(-|/)\d\d\\1\d{4}'
test_string = 'palabra 1 30/06/1970 PALABRA2 12-12-2008 palabra3'
```

```
for match inre.finditer(pattern, test_string):
    print(match)
    print(match.start())
    print(match.end())
    print(match.span())
    print(match.group(0))
```

En el ejemplo anterior se muestra:

```
<re.Match object; span=(10, 20), match='30/06/1970'>
10
20
(10, 20)
30/06/1970
<re.Match object; span=(30, 40), match='12-12-2008'>
30
40
```

```
(30, 40)
12-12-2008
```

## La función findall()

La función findall() devuelve una lista con las ocurrencias en la string de búsqueda correspondientes a todos los grupos del patrón.

```
import re
```

```
pattern = '\d\d-\d\d-\d{4}'
test_string = 'palabra1 30-06-1970 PALABRA2 12-12-2008 palabra3'
```

```
for occurrence inre.findall(pattern, test_string):
    print(occurrence)
```

En este primer ejemplo no hemos introducido grupos en el patrón. El bucle itera, mostrando una ocurrencia cada vez, sobre la lista devuelta por la llamada a findall(), que es la siguiente.

```
['30-06-1970', '12-12-2008']
```

Hay que resaltar que la función findall() devuelve **los grupos** encontrados en cada ocurrencia. En el ejemplo anterior devuelve una lista de strings, cada una de las cuales es una ocurrencia completa correspondiente al grupo 0 (patrón completo). No obstante, si cambiamos el patrón por

```
'((\d\d)-(\d\d)-(\d{4}))'
```

lo que obtenemos es

```
[('30-06-1970', '30', '06', '1970'), ('12-12-2008', '12', '12', '2008')]
```

O sea, una lista de tuplas en la que cada una representa una ocurrencia y cada string corresponde a un grupo en esa ocurrencia. En el ejemplo, la primera string de cada tupla es la ocurrencia completa gracias a que hemos encerrado todo el patrón entre paréntesis, pues el grupo 0 solo se incluye (en lugar de la tupla) si, como en el primer ejemplo, no hay grupos definidos en el patrón. Por ello, si quitáramos los paréntesis más externos obtendríamos:

```
[('30', '06', '1970'), ('12', '12', '2008')]
```

## La función split()

La función split() divide la string de búsqueda en una lista de substrings usando las ocurrencias del patrón como separadores:

```
import re
pattern = '\d\d-\d\d-\d{4}'
test_string = 'palabra1 30-06-1970 PALABRA2 12-12-2008 palabra3'
print(re.split(pattern, test_string))
```

El ejemplo anterior muestra la lista:

```
['palabra1 ', ' PALABRA2 ', ' palabra3']
```

Como con findall(), hay que tener cuidado con la definición de grupos, ya que de haberlos se devuelven también. Si en el ejemplo anterior cambiamos el patrón por

```
'(\d\d)-\d\d-(\d{4})'
```

obtenemos

```
['palabra1 ', '30', '1970', ' PALABRA2 ', '12', '2008', ' palabra3']
```

Nótese que no aparece el mes de la fecha porque no se ha definido grupo para él.

## La función sub()

La función **sub()** sirve para sustituir las ocurrencias del patrón en la string de búsqueda por un nuevo valor:

```
import re
```

```
pattern = '(\d\d)-(\d\d)-(\d{4})'
test_string = 'palabra1 30-06-1970 PALABRA2 12-10-2008 palabra3'
new_string =re.sub(pattern, '<DATE>', test_string)
print(new_string)
```

El resultado del ejemplo anterior es:

```
"palabra1 <DATE> PALABRA2 <DATE> palabra3"
```

La string con el nuevo valor puede contener referencias a grupos de la ocurrencia que vamos a sustituir:

```
import re
```

```
pattern = '(\d\d)-(\d\d)-(\d{4})'
test_string = 'palabra1 30-06-1970 PALABRA2 12-10-2008 palabra3'
```

```
new_string =re.sub(pattern, '\\2/\\1/\\3', test_string)
print(new_string)
```

El resultado del ejemplo es:

```
"palabra1 06/30/1970 PALABRA2 10/12/2008 palabra3"
```

Con ello hemos conseguido intercambiar los grupos 1 y 2 de cada fecha, y sustituir los guiones por barras.

Existe una función parecida llamada **subn()** que, en lugar de devolver la string modificada, devuelve una tupla en la que el primer elemento es la string modificada y el segundo el número de ocurrencias que se han sustituido.

## Flags

A todas las funciones de manejo de expresiones regulares se les pueden añadir como parámetros ciertos flags que modifican su comportamiento. Por ejemplo, el siguiente código:

```
import re
pattern = 'bra\d'
test_string = 'palabra1 30-06-1970 PALABRA2 12-10-2008 palabra3'
print(re.findall(pattern, test_string))
```

da como resultado:

```
['bra1', 'bra3']
```

Pero si añadimos un parámetro con el flag re.IGNORECASE:

```
import re
pattern = 'bra\d'
test_string = 'palabra1 30-06-1970 PALABRA2 12-10-2008 palabra3'
print(re.findall(pattern, test_string,re.IGNORECASE))
```

se obtiene:

```
['bra1', 'BRA2', 'bra3']
```

pues la búsqueda se realiza ignorando la diferencia entre mayúsculas y minúsculas.

Existen 6 posibles flags, cada uno con una versión larga y una abreviada, de una sola letra:

**ASCII, A**

Hace que las secuencias de escape \w, \b, \s y \d busquen sólo caracteres ASCII

**DOTALL, S**

Hace que el metacarácter '.' incluya los caracteres de salto de línea ('\n'). Por omisión el '.' significa "cualquier carácter que no sea un salto de línea".

**IGNORECASE, I**

Ignora la diferencia entre mayúsculas y minúsculas

**LOCALE, L**

Tiene en cuenta las convenciones locales del lenguaje.

**MULTILINE, M**

Búsqueda en strings multilínea. Hace que los metacaracteres ^ y $ coincidan, respectivamente, con el comienzo y fin de cada línea en una string multilínea. Por defecto, coinciden con el comienzo y fin de la string. Por ejemplo, el siguiente código:

```
import re
pattern = "^a\w+"
test_string = """amor
agua
casa
azúcar
"""
print(re.findall(pattern, test_string))
```

da como resultado ['amor'], pero si añadimos el flag multiline:

```
print(re.findall(pattern, test_string, re.M))
```

obtenemos: ['amor', 'agua', 'azúcar']

**VERBOSE, X** (por 'extended')

Facilita usar expresiones más legibles, permitiendo usar espacios para darles formato e introducir comentarios para explicarlas. Por ejemplo:

```
pattern = """
 &[#]              # Comienzo de unreferencia numérica
 (
    0[0-7]+        #Formato octal
  | [0-9]+         #Formato decimal
  | x[0-9a-fA-F]+   #FormatoHexadecimal
 )
 ;                # Punto y coma final
"""
```

en vez de:

```
pattern = "&[#](0[0-7]+|[0-9]+|x[0-9a-fA-F]+);"
```

Si se quieren usar varios flags, se pueden combinar con el operador '|':

```
print(re.findall(pattern, test_string,re.IGNORECASE | re.DOTALL | re.VERBOSE))
```

# Funciones recursivas

## Funciones recursivas (1)

Supongamos que queremos resolver el siguiente problema:

---

Desarrolle una función llamada **sum_ints** que tome como parámetro una lista con valores de diferentes tipos y devuelva la suma de los elementos de tipo *int* que haya en dicha lista.

Por ejemplo:

Recibe: [10,  "Pedro", 42, "Margarita", 18.5, 8]

Devuelve: 10 + 42 + 8 = 60

---

Una posible solución es:

def sum_ints(the_list):

```
    """Suma los númerosenteros de una lista que
      contiene elementos de diferentes tipos.
    """
    result = 0

    foritem in the_list:
        if type(item) == int:
            result +=item

    return result
```

## Funciones recursivas (2)

Supongamos que, en el problema de sumar todos los números enteros de una lista que contiene elementos de diferentes tipos, algunos de esos elementos son, a su vez, listas:

```
lista = [10,  [25, 10, "Pedro"], 42, "Margarita", [12, "casa", 10], 8]
```

y que queremos incluir en la suma los números enteros que pudiera haber en las listas que forman parte de la lista original. En nuestra versión inicial de la solución, tratábamos los elementos que eran de tipo *int* e ignorábamos el resto. Sólo tenemos que modificarla para tratar las listas que nos encontremos, sumando los números enteros que contengan:

```
def sum_ints(the_list):
    """Suma los númerosenteros de una estructura que
       contiene elementos de diferentes tipos
    """
    result = 0

    foritem in the_list:
        if type(item) == int:
            result +=item
        elif type(item) == list:
            result += sum_ints(item)

    return result
```

Básicamente, lo que tenemos es que, si el elemento es de tipo *int*, lo sumamos al resultado, y, si el elemento es una lista, sumamos al resultado la suma de los números enteros que contenga. ¿Y cómo hacemos para calcular esa suma? Simplemente, aplicamos a la lista contenida el mismo algoritmo que estamos aplicando a la lista contenedora (el de la función *sum_ints*).

Cuando resolvemos una parte de un problema aplicando el mismo algoritmo que estamos usando para resolver el problema completo, decimos que estamos aplicando una **solución recursiva**.

Nótese, que tal como está planteada, la solución funciona también cuando algunas de las listas contenidas en la inicial tienen elementos que son, a su vez, listas, y así hasta cualquier nivel de anidamiento.

## Búsqueda de una solución recursiva: paso 1 - el tamaño del problema

Muchas veces, que apliquemos a un problema una solución recursiva, o no, depende de cómo se mire. Por ejemplo, supongamos que queremos desarrollar una función llamada *contains* a la que se le pase una lista de números y un número y devuelva *True* si el número está contenido en la lista y *False* si no lo está.

Si abordamos el problema con una "mentalidad iterativa", la solución es recorrer la lista comparando cada elemento con el valor buscado hasta que lo encontremos, o se nos acaben los valores con los que comparar:

```
def contains(container, value):
    """Determina si elvalor value está en la lista container."""
    # Recorremos la lista comparando los elementos con elvalor buscado
    foritem in container:
        ifitem == value:
            return True  # Hemos encontrado elvalor buscado
```

```
    # Hemos comparado con todos los elementos de la lista sin encontrar
    # elvalor buscado
    return False
```

Si queremos buscar una solución

recursiva, tenemos que buscar una forma de descomponer el problema consiguiendo que aparezcan subproblemas de la misma naturaleza.

Hay que tener en cuenta que los subproblemas incluidos en un problema tienen necesariamente que ser "más pequeños" que el problema que los incluye.

El primer paso para buscar una solución recursiva a un problema es averiguar qué factores determinan el "tamaño" del problema.

En el caso de nuestro ejemplo, los datos del problema son una lista y un número. Para encontrar si el número está en la lista habrá que irlo comparando con cada elemento de la lista. En el peor caso habrá que compararlo con todos los elementos, y en promedio con la mitad. La probabilidad de realizar más comparaciones aumenta cuanto más larga sea la lista. Eso es cierto independientemente de cuál sea el valor buscado; un valor menor no implica que hagamos menos comparaciones, ni uno mayor que hagamos más (a menos que la lista esté ordenada de forma creciente y lo aprovechemos deteniéndonos, cuando encontramos un valor mayor que el buscado).

En resumen, el valor buscado no influye en el trabajo a realizar, el tamaño de la lista sí lo hace; el tamaño del problema viene determinado, en este caso, por el número de elementos de la lista.

Siempre que estamos trabajando con secuencias (strings, tuplas, listas), en general podemos suponer que el tamaño del problema viene determinado por el número de elementos de la secuencia a tratar: normalmente, tardaremos 10 veces más en recorrer una lista de 1000 elementos que una de 100. Cuando el problema viene definido exclusivamente por datos numéricos, generalmente, números más grandes implican problemas más grandes; por ejemplo, la conocida función factorial se define como:

```
si n = 0, n! = 1
si n > 0, n! = n * (n - 1)!
```

6! > 5! > 4! > 3! > ⋯ en cuanto al trabajo a realizar (p.e. número de multiplicaciones).

## Búsqueda de una solución recursiva: paso 2 - el caso base

Cuando conocemos el factor que determina el tamaño del problema a resolver, podemos conocer cuál es el caso más pequeño del problema. ¿Por qué es esto importante? Podemos suponer que cuesta más solucionar un problema cuanto mayor es, por lo tanto, el caso más pequeño es el más simple de solucionar, ya que no depende de otros subproblemas recursivos de menor tamaño y además típicamente tiene una solución directa. Por ejemplo, en el cálculo del factorial, cuando n = 0 sabemos que su factorial es 1 sin necesidad de ningún cálculo (0 es el número más pequeño del dominio del problema del factorial, que sólo está definido para números enteros no negativos).

En el problema de buscar un valor en una lista, dado que hemos determinado que el tamaño del problema está directamente relacionado con la longitud de la lista, está claro que el caso más pequeño es cuando la lista tiene longitud 0, es decir, está vacía. ¿Y cuál es el resultado de la búsqueda cuando la lista está vacía? Evidentemente, *False*, dado que ningún elemento puede encontrarse en una lista vacía. Por tanto, podemos escribir

```
def contains(container, value):
    """Determina si elvalor value está en la lista container"""
    if len(container) == 0:
        result = False
    else:
# falta resolver el problema para el caso de len(container) > 0

    return result
```

El caso más pequeño de un problema recursivo se conoce como **caso base**, ya que es el que pone fin a la expansión de la recursividad, empezando así a retornar una solución parcial. Sin el caso base, la recursión se convertiría en infinita (teóricamente, en la práctica, nunca alcanzaríamos a calcular completamente el resultado buscado).

## Búsqueda de una solución recursiva: paso 3 - descomposición del problema

Una vez resuelto el caso base, hay que estudiar cómo descomponer el problema general, de modo que obtengamos problemas más pequeños, pero buscando que algunos de estos problemas sean de la misma naturaleza que el original y podamos operar recursivamente sobre ellos (si no tenemos esto en mente, estamos buscando una solución iterativa, no recursiva).

De esta forma, en última instancia, se acabará alcanzando el caso base, que nos proporcionará una solución parcial, a partir de la cual construir la solución del caso general del problema.

Podemos plantearnos que lo que buscamos es quitar uno o más "trozos" del problema, que podamos resolver más o menos directamente, de tal forma que lo que quede una vez quitado ese trozo sean subproblemas de la misma naturaleza que el original, pero más pequeños.

En el ejemplo que estamos desarrollando, ¿qué le podemos quitar a una lista para quedarnos con una lista más pequeña? Normalmente, definimos una lista como una "secuencia de elementos" que es una forma iterativa de verla (los problemas de secuencias se resuelven de forma natural iterando por los elementos de las mismas). Necesitamos una forma distinta de definir una lista. Una posibilidad (entre otras) es decir que una lista puede ser de dos formas, o sea, tenemos dos casos posibles:

- está vacía, o, si no,
- está formada por un elemento seguido por una .

  lista

Si partimos de esta definición de lista podemos, cuando no está vacía, simplemente, separar el primer elemento y lo que nos queda es una lista con el resto de los elementos:

```
    first = list[0]
    tail = list[1:]
```

El primer elemento podemos compararlo con el valor buscado; si coinciden, ya hemos resuelto el problema (se ha encontrado), si no, sólo tenemos que aplicar el mismo algoritmo al resto de la lista que hemos separado (o sea, resolver recursivamente un problema más pequeño) y devolver como resultado lo que nos depare esa búsqueda, sabiendo además, que de esta forma, si no se encontrase el valor buscado, en algún momento del proceso el resto acabará siendo una lista vacía e, igualmente, habremos entonces resuelto el problema (no se habrá encontrado). Por tanto, el **caso general** lo podemos implementar como sigue:

```
    if first == value:
        result = True
    else:
        result = contains(tail, value)
```

Lo cual podemos leer como: "para una lista no vacía (caso general), un valor se encuentra en ella si es igual al primero de la lista o, si no, si se encuenta en el resto de la lista".

Así pues, la solución completa, combinando casos base y general, quedaría como sigue (tras eliminar las variables *first* y *tail*, que no usamos más que una vez).

```
def contains(container, value):
    """Determina si elvalor value está en la lista container."""
    if len(container) == 0:
        result = False
    else:
        if container[0] == value:
            result = True
        else:
            result = contains(container[1:], value)
```

```
    return result
```

## Esquema de una solución recursiva

Ahora tenemos dos formas de resolver el problema de encontrar un valor en una lista:

| Solución iterativa | Solución recursiva |
| --- | --- |
| def contains(container, value): result = False for item in container: if item == value: result = True break return result | def contains2(container, value): if len(container) == 0: result = False else: if list[0] == value: result = True else: |

| | |
|---|---|
| | result = contains2(list[1:], value)<br>return result |

La solución iterativa sigue el conocido esquema de recorrido de una secuencia. La solución recursiva muestra el esquema habitual de una solución de esta clase. La recursividad siempre divide el dominio del problema al menos en dos partes: caso base/caso general, por lo que en el algoritmo de la solución siempre estará presente una estructura de selección (*if*) para discriminar entre ambas partes. Podemos observar cómo ocurre esto igualmente en otros problemas clásicos:

| Factorial de un número natural | Solución recursiva |
|---|---|
| si n = 0, n! = 1<br><br>si n > 0, n! = n * (n - 1)! | def factorial(n):<br>if n == 0:<br>result = 1<br>else:<br>result = n * factorial(n - 1)<br><br>return result |

| n-ésimo número de Fibonacci | Solución recursiva |
|---|---|
| si n = 0, fibo(n) = 0<br><br>si n = 1, fibo(n) = 1<br><br>si n > 1, fibo(n) = fibo(n - 1) + fibo(n - 2) | def fibo(n):<br>if n <= 1:<br>result = n<br>else:<br>result = fibo(n - 1) + fibo(n - 2)<br>return result |

No obstante lo dicho, una vez que adquiramos soltura en su realización, soluciones recursivas como la mostrada al principio son simplificables obviando la variable result, al igual que ya sabemos hacer con la solución iterativa mostrada también al principio.

# Ficheros de texto (1)

## Archivos de texto

Las aplicaciones del mundo real procesan (leen o/y producen) diversos datos que se guardan en archivos externos. Muchos de ellos son textos. Ejemplos de archivos que tienen un formato de texto incluyen aquellos con extensión .txt, .csv, .html ... ¡y, desde luego, .py!

El contenido de un archivo de texto es una secuencia de **caracteres**(a), incluidos caracteres terminadores de línea como *newline* '\n'.

Podemos también considerar un archivo de texto como una secuencia de **líneas**, siendo cada línea una secuencia de caracteres terminada con '\n'. Un archivo de texto puede contener líneas **vacías**, esto es, que sólo tienen el carácter '\n'. (b)

Trabajar con archivos en general comprende **tres etapas**:

1. **Abrir** el archivo (se establece una conexión entre el programa y el archivo físico en el sistema de ficheros del ordenador).
2. **Leer** de, **o escribir** en, el archivo (se envían datos desde el archivo al programa, o desde el programa al archivo).
3. **Cerrar** el archivo (se libera la conexión entre el programa y el archivo físico en el sistema de ficheros; lo que pudiera requerir que previamente se completen operaciones de escritura aún pendientes).

(a) Todo archivo, del tipo que sea, hoy en día es una secuencia de octetos, o sea, de bloques de 8 bits, cuyas combinaciones de 0 y 1 codifican, de acuerdo con algún esquema preestablecido (que puede ser octeto a octeto o en grupos de octetos), algún tipo de dato. Así pues, cuando decimos que un archivo de texto contiene una secuencia de *caracteres*, lo que estamos queriendo decir es que cuando se construye el archivo los octetos que lo componen se escriben utilizando alguna codificación [*encoding*] de caracteres definida en un *estándar* o *norma*, por lo que, naturalmente, al leerlos posteriormente debieran ser interpretados de acuerdo con dicha codificación. Estándares de codificación de caracteres hay varios, no obstante, en nuestras prácticas escribiremos y leeremos usando una misma codificación, por lo que no nos afectará.
(b) Pudiéramos encontrarnos (aunque no sea recomendable según el caso) con que la última línea no esté vacía y no termine en '\n'. O incluso que el fichero esté vacío (contenga cero caracteres, ni siquiera un '\n'). Estrictamente, un fichero que contuviera un solo carácter que fuera un '\n' no estaría vacío, pues contendría una línea, eso sí, vacía.

## Apertura de un archivo

Para abrir un archivo de texto en Python llamamos a la función estándar **open**():

```
f = open('data.txt', 'r')
```

La función open() tiene, generalmente, dos parámetros de tipo string:

- El nombre del archivo que se quiere abrir y
- La  en caso de abrir archivo para su lectura ().

  'r'

  read

Podemos especificar una ruta absoluta o relativa (en el sistema de ficheros) en la primera string. En el ejemplo hemos asumido que el archivo de entrada y la secuencia de comandos de Python (el programa) están en el mismo directorio.

La función open() devuelve una referencia al archivo abierto. En otras palabras: el flujo (de entrada, en este caso) se ha abierto y la variable de archivo, que hemos llamado f, a la izquierda de la asignación nos proporcionará la conexión, más adelante necesaria.

Si no hay en el disco un archivo con el nombre especificado, se genera un error (FileNotFoundError).

En un tercer argumento opcional podemos especificar la codificación a emplear en el archivo, por ejemplo:

```
f = open('data.txt', 'r', encoding = 'utf-8')
```

## Tratamiento de un archivo de texto

Supongamos que el archivo poem.txt contiene el texto mostrado a continuación:

*Twinkle, twinkle, little star,how I wonder what you are.Up above the world so high,like a diamond in the sky.*

Para leerlo y mostrarlo en pantalla escribiríamos lo siguiente:

```
f = open('poem.txt', 'r')
print(f.readline())       # 'Twinkle, twinkle, little star,\n'
```

Con el método **readline**(), leemos la primera línea del archivo. Este método lee y devuelve una línea, incluyendo el carácter de final de línea.

En la variable de archivo f se actualiza la posición real hasta la que se ha avanzado en un archivo después de cada lectura, por lo que la siguiente llamada devolverá la segunda línea:

```
print (f.readline())     # 'how I wonder what you are.\n'
```

Después de leer todas las líneas, la posición real llega al final del archivo y, en adelante, el método readline() devuelve '' (una string vacía).

Para leer todas las líneas, a menudo usamos la instrucción while:

```
f = open('poem.txt', 'r')
line = f.readline()
```

```
while line != '':
    print(line, end = '') # la línea ya contiene un '\n' (salvo quizá la última)
    line = f.readline()
```

También es posible la siguiente versión (una string vacía se interpreta como False en una condición; recuérdese que una línea vacía no es una string vacía):

```
f = open('poem.txt', 'r')
line = f.readline()
```

```
while line:
    print(line, end = '') # la línea ya contiene un '\n' etc.
```

```
    line = f.readline()
```

Después de leer todos los datos necesarios, el archivo debe cerrarse inmediatamente:

```
f.close()
```

Después del cierre, se liberan todos los recursos del programa relacionados con el archivo, y el archivo queda libre para poder ser usado por otros programas.

Al leer líneas de un archivo, puede ser interesante ver su contenido real (todos los caracteres, incluido el final de las líneas). La función **repr**() es útil para este propósito, pues nos muestra los valores en un formato que podríamos usar para introducirlos como literales en nuestro código Python:

```
f = open('poem.txt','r')
line = f.readline()
```

```
while line != '':
    print(repr(line))
    line = f.readline()

f.close()
```

La salida (obsérvese que incluye los delimitadores ' de literal de string):

'Twinkle, twinkle, little star,\n''How I wonder what you are.\n''Up above the world so high, \t\n''Like a diamond in the sky.\n'Observe que la tercera línea contiene más espacios y un tabulador antes del \n final, que con un simple print no se podrían apreciar.

En Python, caracteres como el espacio, el tabulador y el final de línea se consideran **espaciadores** (*whitespaces*). A veces, debemos eliminar todos los espaciadores iniciales y finales antes de procesar una línea. Podemos usar el método **strip**() para esto, pues devuelve una copia de la string sobre la que sea aplica, pero sin los espaciadores al principio y al final que esta pudiera tener:

```
f = open('poem.txt','r')
line = f.readline()
```

```
while line != '':
    print(repr(line.strip()))
    line = f.readline()
```

```
f.close()
```

La salida:

'Twinkle, twinkle, little star,''How I wonder what you are.''Up above the world so high,''Like a diamond in the sky.'

Obsérvese que en la instrucción print no se usa en esta ocasión el parámetro end = '', sino que se usa el terminador por omisión (\n), ya que el carácter \n ha sido eliminado por la operación strip().

El método strip() puede tener un parámetro opcional: una string indicando el repertorio de caracteres que se quiere eliminar (no necesariamente espaciadores) de principio y final. También están disponibles las variantes **lstrip**(), que elimina los espaciadores, o caracteres especificados, del principio (***l****eft*) de una string y **rstrip**() que hace lo mismo por el final (***r****ight*). Repasar la semana 9 "Más sobre strings".

### Tratamiento de un archivo usando la sentencia for

La sentencia for puede ser muy útil al procesar archivos de texto.

Si conociéramos de antemano la cantidad de líneas (algo inusual), en el ejemplo podríamos repetir la llamada readline() para imprimir las 4 líneas del poema:

```
f = open('poem.txt', 'r')
```

```
for i in range(4):    print(f.readline(), end='')
```

```
f.close()
```

Pero sobre todo puede usarse, facilitando además el ciclo de lectura, en caso de un número desconocido de
líneas (que es lo más usual):

```
f = open('poem.txt', 'r')
```

```
for line in f :    print(line, end='')
```

```
f.close()
```

# Ficheros de texto (2)

### Leer un fichero en una única string

A veces, necesitamos leer el contenido de un archivo de entrada como una sola string:

```
f = open('poem.txt', 'r')
text = f.read()
print(len(text))          #  totalnumber of characters
```

```
print(text)             #  contents of the file
f.close()
```

El método **read**() puede tener como parámetro el número de caracteres a leer:

```
f = open('poem.txt','r')
print(f.read(1))          # first character
print(f.read(5))          # next 5 characters
print(f.read())           # therest of the file's contents
f.close()
```

## Escribir en un fichero de texto

Al abrir el fichero con open() especificaremos el modo de escritura mediante el carácter **'w'** como segundo parámetro. Si el archivo no existe, se creará vacío, y aun cuando existiera previamente, su contenido será sobreescrito.

Una vez abierto, podemos escribir usando el método **write()**, cuyo parámetro ha de ser una **string**. Al usarlo, deberemos incluir explícitamente todos los caracteres de fin de línea que queramos contenga el fichero de salida:

```
f = open('output.txt','w')
f.write('We are learning\n')
f.write('about using text files\n')
f.write('in Python\n')
f.close()>
```

Las mismas tres líneas de texto se podrían igualmente haber enviado al archivo llamando al método write() una sola vez:

```
f = open('output.txt', 'w')
f.write('We are learning\nabout using text files\nin Python\n')
f.close()
```

Hay que **acordarse de cerrar siempre los archivos abiertos**. En caso de no cerrarse, no se puede estar seguro de si lo que hemos mandado escribir realmente acaba guardado en el disco.

También se puede escribir en un fichero de texto usando la función **print**(). Podemos redireccionar la salida usando un parámetro opcional para escribir en un fichero, en vez de en la salida estándar:

```
f = open('primes.txt', 'w')
fornumber in (2, 3, 5, 7, 11, 13, 17, 19):
```

```
    print(number, end = ' ', file = f)
f.close()
```

El archivo primes.txt contendrá una sola línea conteniendo varios números separados por un espacio:

```
2 3 5 7 11 13 17 19
```

Si no se hubiese especificado el parámetro end cada número estaría en una línea diferente.

Obsérvese que mientras que write() requiere que el parámetro sea string, print() acepta **otros tipos** convirtiéndolos a string antes de escribir, ya sea en la terminal, como sabemos, o en el fichero de texto que se indique.

También se puede escribir en un fichero de texto usando la función **print**(). Podemos redireccionar la salida usando un parámetro opcional para escribir en un fichero, en vez de en la salida estándar:

```
f = open('primes.txt', 'w')
fornumber in (2, 3, 5, 7, 11, 13, 17, 19):
    print(number, end = ' ', file = f)
f.close()
```

El archivo primes.txt contendrá una sola línea conteniendo varios números separados por un espacio:

```
2 3 5 7 11 13 17 19
```

Si no se hubiese especificado el parámetro end cada número estaría en una línea diferente.

Obsérvese que mientras que write() requiere que el parámetro sea string, print() acepta **otros tipos** convirtiéndolos a string antes de escribir, ya sea en la terminal, como sabemos, o en el fichero de texto que se indique.

## La cláusula with con ficheros

Cuando se trabaja con archivos, se recomienda usar la cláusula **with**:

```
withopen('poem.txt','r', encoding='utf-8')asf:
forlineinf:
        print(line, end='')
```

En el ejemplo anterior, el archivo abierto resulta referencia do con la variable **f**. Al salir del bloque with(ya sea porque se alcanza el final normalmente o cualquier otro motivo, como p.e. la ejecución en su interior de break o return) el archivo se **cierra automáticamente**.

Veamos cómo copiar con with el contenido de un archivo. El primer archivo debe abrirse para leer, el segundo para escribir:

```
withopen('original.txt', 'r')asfr:
withopen('copy.txt', 'w')asfw:
      fw.write(fr.read())
```

También podríamos haber enumerado en un solo with los archivos que se manejan:

```
withopen('original.txt', 'r')asfr, open('copy.txt', 'w')asfw:
   fw.write(fr.read())
```

### Adición de contenido a un archivo existente

Además de los modos de lectura y escritura, también es posible agregar contenido a un archivo.

Supongamos que las siguientes 3 líneas (todas terminadas en final de línea) constituyen el contenido de un cierto archivo llamado *names.txt*:

John

Paul

George

Abriremos este archivo en el modo ***append*** (añadir) usando **'a'** como segundo parámetro:

```
file = open('names.txt', 'a')
file.write('Ringo\n')
file.close()
```

Al abrir el archivo en modo añadir, la posición de escritura se coloca al final del mismo. En el ejemplo, a partir de ahí se le añade una línea, que será la 4ª del archivo; y finalmente se cierra.

El contenido del archivo *names.txt* resulta:

John

Paul

George

Ringo

# Terminología Básica de BBDD

Los **datos** son propiedades de los objetos/entidades, normalmente obtenidos por medición u observación. Para que los datos sean procesados, deben ser expresados. Los datos se pueden expresar por texto, voz, imagen (gráficamente), electrónicamente, etc. Cuando lo simplificamos, podemos decir que los datos se expresan mediante signos o señales.

Para trabajar con los datos, estos se agrupan en unidades lógicas superiores denominadas **registros** (oraciones). Un registro es una unidad de datos lógica. Sin embargo,

un registro no es el elemento de datos más pequeño, un registro (oración) puede ser descompuesto. Al igual que en la vida cotidiana, la oración consta de palabras, y los datos dividen el registro en **atributos**. El atributo es la parte más pequeña direccionable de la oración. Igual que la palabra se puede descomponer en letras individuales, en algunos casos es posible dividir un atributo. Sin embargo, igual que las palabras, perdería su significado.

Los atributos pueden ser atómicos o estructurados. Un ejemplo de un atributo estructurado puede ser una dirección (calle, número de casa, ciudad, código postal, ...). Es bueno evitar tales atributos y dividirlos en atributos atómicos. Los atributos tienen una cierta posición en la oración (registro), y tienen su significado. Para que un atributo desempeñe su función correctamente, los valores que adquirirá deben ser significativos. El conjunto de valores permitidos que un atributo puede adquirir se denomina **dominio**.

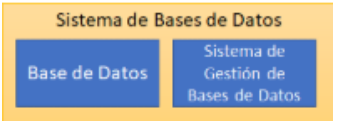
El dominio no sólo especifica que el atributo NOMBRE DE UNA PERSONA es una cadena de caracteres. Esta es la determinación del tipo de datos (se explicará más adelante con más detalle). El dominio es más específico. Representa todos los valores significativos para el atributo dado. En el dominio del atributo NOMBRE DE UNA PERSONA se puede incluir el valor "Juan", pero el valor "x7br_15" no cae en el dominio de este atributo, incluso siendo una cadena de texto.

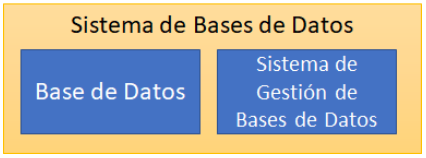
El tipo de registro determina qué atributos (incluidos los dominios) tiene el registro. El orden de los atributos individuales también juega un papel importante. El tipo de registro (oración) se especifica por sus atributos, por ejemplo, NOMBRE DE UNA PERSONA, ACTIVIDAD, OBJETO. Una frase de este tipo puede ser: "Juan conduce un coche".

La información es intangible. La información son datos que tienen significado. La información es, por lo tanto, un subconjunto de datos. La información puede responder a preguntas, reduciendo así la ignorancia (incertidumbre). La información puede estar contenida tanto en los signos (o señales) como en su disposición. La apariencia física puede variar (texto, imagen, señales, etc.).

Los datos proporcionan información sólo a aquellos que los entienden, que pueden reconocer su sintaxis y entender la semántica. Lo que es información para alguien, puede ser sólo datos para otro.

Los registros del mismo tipo se agrupan en **archivos de datos**. Para trabajar con ellos (por ejemplo, buscar, eliminar, editar), cada registro debe distinguirse claramente de los demás. Por lo tanto, cada registro debe ser identificado por la clave de archivo. Una clave es un conjunto de atributos que identifican de forma única un registro. El número de atributos que pertenecen a la clave se llama *k*. El número total de atributos de la oración se denota como *n*. Siempre se cumple que $k \leq n$. El objetivo es mantenerlo lo más pequeño posible. Todas las claves mínimas crean un espacio *K **. Una de las claves está marcada como la **clave principal**.






Como puede verse en la imagen, un **Sistema de Base de Datos** consta de dos componentes básicos:

- La Base de Datos.
- Un Sistema de Gestión de Bases de Datos.

Una **Base de Datos** es un conjunto de archivos homogéneos y estructurados que almacenan datos para su posterior procesamiento. La distribución de los datos en los archivos individuales responde a un significado global. Todos los datos dentro de un mismo archivo tienen la misma estructura.

Para acceder a la base de datos, existe una herramienta llamada **Sistema de Gestión de Base de Datos**. A menudo, también se utiliza la abreviatura DBMS (por Data Base Management System). Un sistema de administración de base de datos es una herramienta de software integrada que permite definir, crear y administrar el acceso a la base de datos y trabajar con ella. Normalmente, es una colección de programas que conforman la interfaz entre los programas de aplicación y los datos almacenados.

El Sistema de Gestión de Bases de Datos tiene muchas funciones: en primer lugar, permite manipular los datos; asegura que sólo los usuarios autorizados puedan acceder a los datos; también permite el acceso concurrente de múltiples usuarios; proporciona gestión de transacciones; dependiendo del modelo utilizado, crea una base de datos y define su esquema (estructura); en caso de fallo, permite la recuperación; y comprueba la integridad de los datos, entre otras muchas tareas.

Los beneficios que aporta el DBMS al acceso directo a los datos son indiscutibles. Los principales incluyen:

- **Abstracción de datos**: el usuario no trabaja directamente con los archivos de origen, sino con estructuras formalizadas en el nivel lógico superior de abstracción.
- **Independencia de los datos**: si los datos físicos cambian, no afecta el trabajo de los programas de aplicación. La interfaz de datos seguirá siendo la misma, cara al exterior.
- **Administración de datos centralizada**: todos los datos están en un solo lugar. Todo es tratado de manera similar. Es posible mostrar una descripción de la estructura de datos.
- **La capacidad de formular consultas ad hoc fuera de los programas de aplicación**: los usuarios pueden hacer consultas aleatoriamente en una base de datos a través del DBMS. No necesitan otros programas para hacerlo.

# SQL y Python

## ¿Qué es SQL?

SQL es un lenguaje estándar para acceder y manipular bases de datos. SQL se convirtió en un estándar del American National Standards Institute (ANSI) en 1986 y de la Organización Internacional de Normalización (ISO) en 1987.

Las siglas SQL significan Lenguaje de Consulta Estructurado (**S**tructured **Q**uery **L**anguage).

Usando el lenguaje SQL se pueden crear nuevas bases de datos, crear tablas, ejecutar consultas, recuperar datos, insertar, actualizar y eliminar registros, etc.

Existen diferentes versiones del lenguaje SQL pero, para cumplir con el estándar, todos admiten al menos los comandos principales.

Python puede usarse para desarrollar programas que manipulen bases de datos. Estos programas se conectan con un Sistema de Gestión de Bases de Datos (DBMS, en inglés), usando un *conector*, e interactúan con las bases de datos por medio de *comandos* SQL.

En este tutorial vamos a usar SQLite 3 como Sistema de Gestión de Bases de Datos. Para interactuar con SQLite 3 desde un programa escrito en Python, debemos importar el módulo *sqlite3*:

```
import sqlite3
```

## Crear y eliminar BBDD

La manipulación de bases de datos usando el lenguaje SQL se realiza mediante sentencias que se componen de una o varias instrucciones.

Para crear una base de datos se usa el comando

**CREATE DATABASE *nombre*;**

Por ejemplo, la siguiente sentencia crea una base de datos llamada *FormulaUno*:

```
CREATE DATABASE FormulaUno;
```

Aunque, por convención, en este tutorial pondremos en mayúscula las instrucciones SQL, el lenguaje no hace distinción entre mayúsculas y minúsculas. Obsérvese que la sentencia acaba en punto y coma, esto es obligatorio en algunos SGBD y no en otros, pero es el estándar para **separar** las sentencias SQL cuando queremos indicar una secuencia de ellas.

Desde Python conectamos con una base de datos SQLite usando la función **connect**:

```
db = sqlite3.connect("FormulaUno")
```

Si la base de datos existe, la función connect abre una *conexión* con la misma y la devuelve, quedando en el ejemplo asignada a la variable db para su uso posterior. Si la base de datos no existe, la crea primero y luego abre la conexión. Si no puede crearla se produce un error.

Cuando terminemos de trabajar con una base de datos debemos cerrar la conexión correspondiente:

```
db.close()
```

También se puede establecer la conexión usando una cláusula de contexto (with), de esta manera, la propia cláusula de contexto se encargará de cerrar la conexión cuando termine su ámbito, evitando que tengamos que escribir la instrucción de cierre.

```
with sqlite3.connect("FormulaUno") as db:
<Operaciones con la base de datos>
```

No existe un comando para eliminar una base de datos en *sqlite3*. Como la base de datos se almacena en un fichero con la**extensión db** (p.e.FormulaUno.db), basta con eliminar ese fichero. En el estándar SQL se usa el comando DROP DATABASE nombre;. Por ejemplo:

```
DROP DATABASE FormulaUno;
```

## Creación de tablas

Para crear una tabla en la base de datos activa, se usa el comando:

**CREATE TABLE *table_name* (*columna1 tipo1*, *columna2 tipo2, columna3 tipo3*, ...)**

Por ejemplo:

```
CREATE TABLE Pilotos (
   id INTEGER,
   Piloto TEXT(20),
   Escudería TEXT(20),
   País TEXT(20),
   Puntos INTEGER,
   Victorias INTEGER
)
```

Crea una tabla cuyos registros tienen la estructura:

| id | Piloto | Escudería | País | Puntos | Victorias |
|---|---|---|---|---|---|

En un entorno interactivo proporcionado por un SGBD, los comando SQL se escriben tal cual. Para usarlos desde Python, se debe crear un **cursor** asociado a la base de datos:

```
db = sqlite3.connect("FormulaUno")
cursor = db.cursor()
```

Una vez creado el cursor, se usa para ejecutar los comandos SQL, que se le pasan como parámetros de tipo string:

```
sql = """
   CREATE TABLE Pilotos (
      id INTEGER, Piloto TEXT(20),
      Escudería TEXT(20),
      País TEXT(20),
      Puntos INTEGER,
      Victorias INTEGER
   );
"""
cursor.execute(sql)
```

En la creación de la tabla se usan los tipos de datos INTEGER y TEXT. SQLite usa los tipos de datos: NULL, INTEGER, REAL, NUMERIC Y BLOB; otros SGBD pueden ofertar una variedad de tipos más amplia. La siguiente tabla muestra la equivalencia entre los tipos de Python y los de SQLite:

| Python type | SQLite type |
|---|---|
| None | NULL |
| int | INTEGER |
| long | INTEGER |
| float | REAL |
| str (UTF8-encoded) | TEXT |
| unicode | TEXT |
| buffer | BLOB |

Para **eliminar** una tabla se usa el comando

**DROP TABLE *table_name***

Por ejemplo:

```
DROP TABLEPilotos;
```

## La instrucción INSERT INTO

Para añadir registros en una tabla de la base de datos se usa la instrucción

**INSERT INTO *Tabla* (*columna1*, *columna2*, ...) VALUES (*valor1*, *valor2*, ...)**

Por ejemplo:

```
INSERT INTO Pilotos VALUES (1, 'Hamilton', 'Mercedes', 'Inglaterra', 250, 8)
```

En Python:

```
sql = "INSERT INTO Pilotos VALUES (?, ?, ?, ?, ?, ?)"
values = (1, 'Hamilton', 'Mercedes', 'Inglaterra', 250, 8)
cursor.execute(sql, values)
```

Nótese, que cuando se usa SQL desde un programa (en cualquier lenguaje, no sólo Python), **no se deben incluir los valores** de los campos en la string del comando SQL, sino que deben proporcionarse como una tupla separada, sustituyéndolos por interrogaciones en la sentencia SQL; esto es por razones de seguridad, ya que si no, se posibilitaría una intrusión no autorizada a la base de datos mediante "inyección de código".

Existe la opción de no proporcionar valores para todas las columnas, en cuyo caso hay que especificar en el comando INSERT INTO las columnas para las que se proporcionan valores:

```
sql = "INSERT INTO Pilotos (id, Piloto, Escudería, Puntos) VALUES (?, ?, ?, ?)"
values = (2, 'Bottas', 'Mercedes', 188)
cursor.execute(sql, values)
```

En el ejemplo anterior, se omiten las columnas *País* y *Victorias*, que quedan almacenadas en la base de datos con el valor *NULL*, equivalente al valor *None* de Python.

## La instrucción SELECT

Para recuperar información de una base de datos se usa la instrucción

**SELECT *columna1*, *columna2*, ... FROM *Tabla***

Por ejemplo:

```
SELECT Piloto, Puntos FROM Pilotos
```

El resultado de una instrucción SELECT se almacena en una tabla temporal, el *result_set*, que, en un entorno interactivo, normalmente se muestra al usuario, o bien se usa como parte de otro comando. Para ejecutar la instrucción SELECT desde Python hay que recurrir al consabido cursor:

```
sql = "SELECT Piloto, Puntos FROM Pilotos"
cursor.execute(sql)
```

Para acceder luego, desde Python, al resultado de la sentencia SELECT, se debe ejecutar el método **fetchall()** del cursor, lo que proporciona una lista de tuplas, cada una de las cuales representa un registro del *result_set*:

```
result_set = cursor.fetchall()
foritem in result_set:
    print(item)
```

Suponiendo la siguiente tabla *Pilotos*:

| id | Piloto | Escudería | País | Puntos | Victorias |
|---|---|---|---|---|---|
| 1 | L. Hamilton | Mercedes | Inglaterra | 250 | 8 |
| 2 | V. Bottas | Mercedes | Finlandia | 188 | 2 |
| 3 | M Verstappen | Red Bull | Holanda | 181 | 2 |
| 4 | S. Vettel | Ferrari | Alemania | 156 | 0 |
| 5 | C. Leclerc | Ferrari | Mónaco | 132 | 0 |
| 6 | P. Gasly | Red Bull | Francia | 63 | 0 |
| 7 | C. Sainz Jr. | McLaren | España | 58 | 0 |
| 8 | K, Raikkonen | Alfa Romeo | Finlandia | 31 | 0 |
| 9 | D. Kvyat | Toro Rosso | Rusia | 27 | 0 |
| 10 | L. Norris | McLaren | Inglaterra | 24 | 0 |

El resultado del ejemplo anterior sería:

```
('Hamilton', 250)
('Bottas', 188)
('Verstappen', 181)
('Vettel', 156)
('Leclerc', 132)
('Gasly', 63)
('Sainz Jr.', 58)
('Raikkonen', 31)
('Kvyat', 27)
('Norris', 24)
```

Nótese que cada registro del *result_set* es siempre una tupla, aunque tenga un solo elemento; si en el ejemplo anterior cambiamos la sentencia SELECT para obtener sólo los pilotos:

```
sql = "SELECT Piloto FROM Pilotos"
cursor.execute(sql)
result_set = cursor.fetchall()
```

Se obtiene el siguiente valor para *result_set*:

```
[('Hamilton',), ('Bottas',), ('Verstappen',), ('Vettel', ), ('Leclerc',), ('Gasly',), ('Sainz Jr.',), ('Rai
kkonen',), ('Kvyat',), ('Norris',)]
```

El primer piloto no es result_set[0], sino result_set[0][0]

Si se quieren recuperar todas las columnas, basta con poner un **asterisco** en vez de enumerarlas:

```
SELECT * FROM Pilotos;
```

A veces, el resultado de una instrucción SELECT puede tener registros repetidos; por ejemplo:

```
SELECT Escudería FROM Pilotos;
```

da como resultado :

```
[('Mercedes',), ('Mercedes',), ('Red Bull',), ('Red Bull',), ('McLaren',), ('Alfa Romeo',), ('Toro Rosso',), ('McLaren',)]
```

Si queremos evitar las repeticiones, podemos añadir a la instrucción SELECT la cláusula **DISTINCT**:

```
SELECT DISTINCT Escudería FROM Pilotos;
```

y obtenemos:

```
[('Mercedes',), ('Red Bull',), ('McLaren',), ('Alfa Romeo',), ('Toro Rosso',)]
```

## La cláusula WHERE

La cláusula WHERE se usa para establecer condiciones para filtrar los datos de una búsqueda. Por ejemplo, dada la siguiente tabla *Pilotos*:

| id | Piloto | Escudería | País | Puntos | Victorias |
|---|---|---|---|---|---|
| 1 | Hamilton | Mercedes | Inglaterra | 250 | 8 |
| 2 | Bottas | Mercedes | Finlandia | 188 | 2 |
| 3 | Verstappen | Red Bull | Holanda | 181 | 2 |
| 4 | Vettel | Ferrari | Alemania | 156 | 0 |
| 5 | Leclerc | Ferrari | Mónaco | 132 | 0 |
| 6 | Gasly | Red Bull | Francia | 63 | 0 |
| 7 | Sainz Jr. | McLaren | España | 58 | 0 |
| 8 | Raikkonen | Alfa Romeo | Finlandia | 31 | 0 |
| 9 | Kvyat | Toro Rosso | Rusia | 27 | 0 |
| 10 | Norris | McLaren | Inglaterra | 24 | 0 |

El código:

```
sql = "SELECT Piloto, Puntos FROM PilotosWHERE Puntos > 150"
cursor.execute(sql)
select_result = cursor.fetchall()
foritem in select_result:
    print(item)
```

Da como resultado:

```
('Hamilton', 250)
('Bottas', 188)
('Verstappen', 181)
('Vettel', 156)
```

La condición de la cláusula WHERE puede ser compuesta:

```
SELECT Piloto, Puntos FROM Pilotos WHERE Puntos > 150 AND Victorias > 2
```

En las condiciones de una cláusula WHERE se pueden usar:

- Los operadores booleanos (AND, OR, NOT)
- Los operadores relacionales (<, <=, =, <>, !=, >= >)
- Otros operadores (BETWEEN, LIKE, IN)

(los operadores de desigualdad <> y != son equivalentes)

El operador BETWEEN sirve para establecer un rango:

```
SELECT Piloto, Puntos FROM Pilotos WHERE Puntos BETWEEN 150 AND 200
```

El operador LIKE sirve para establecer un patrón con comodines. En SQLite se pueden usar dos comodines, el carácter '%', que significa "cero o más caracteres" y el carácter '_', que significa "un carácter". Por ejemplo:

```
SELECT Piloto, Puntos FROM Pilotos WHERE Piloto LIKE '%n'
```

encuentra todos los registros cuyos pilotos tienen un nombre termina con la letra 'n', precedida de cualquier número de caracteres (Hamilton, Verstappen y Raikkonen).

Mientras que:

```
SELECT Piloto, Puntos FROM Pilotos WHERE País LIKE '%l_n%'
```

Encuentra los registros cuyo campo país contiene una 'l' separada de una 'n' por un único carácter y con cualquier número de caracteres por delante y por detrás de ese trío de letras (Finlandia y Holanda).

Nótese, de este último ejemplo, que los campos que se incluyen en una cláusula WHERE no tienen que ser los mismos que se incluyen en el resultado de la búsqueda.

Por último, el operador IN permite comprobar si un valor está en una "tupla".

```
SELECT Piloto, Puntos FROM Pilotos WHERE País IN ('Alemania', 'Inglaterra')
```

## La cláusula ORDER BY

Los resultados de una consulta se pueden ordenar usando la cláusula

**... ORDER BY *columna1*, *columna2*, ...**

Las opciones **ASC** o **DESC** tras cada columna significan, respectivamente, orden ascendente o descendente para la misma; pero si no se pone nada, se presupone ascendente.

Por ejemplo:

```
sql = "SELECT * FROM Pilotos WHERE Puntos > 150 ORDER BY Escudería DESC, Piloto"
cursor.execute(sql)
select_result = cursor.fetchall()
print(select_result)
```

```
[(3, 'Verstappen', 'Red Bull', 'Holanda', 181, 2), (2, 'Bottas', 'Mercedes', 'Finlandia', 188, 2), (1, 'Hamilton', 'Mercedes', 'Inglaterra', 250, 8)]
```

## Las instrucciones UPDATE y DELETE

Se puede *actualizar* uno (o más) registros —esto es, escribir en uno o más de sus campos— de una tabla usando la instrucción

**UPDATE *tabla* SET *columna1* = *valor1*, *columna2* = *valor2*, ... WHERE *condición***

Por ejemplo:

```
UPDATE Pilotos SET Puntos = 300 WHERE Piloto = 'Hamilton';
```

La cláusula WHERE en la instrucción UPDATE es casi obligatoria; se puede omitir, pero, en ese caso, la actualización ¡afectará a todos los registros!, al no haber ningún filtro.

Lo mismo sucede en el caso de querer eliminar uno (o más) registros de alguna tabla, lo que se hace usando la instrucción

**DELETE FROM *tabla* WHERE *condición***

Por ejemplo:

```
DELETE FROM Pilotos WHERE Escudería = 'Ferrari'
```

¡Olvidar la cláusula WHERE en una instrucción DELETE **vaciaría la tabla** en cuestión!