Tipos de datos estructurados

Ricardo Pérez López

IES Doñana, curso 2022/2023

Generado el 2022/11/14 a las 14:23:00

Índice

1. Introducción

1.1. Composición

Hasta ahora, hemos aprendido que:

- Un programa está compuesto por *instrucciones*.
- Las instrucciones de un programa son las *expresiones* y las *sentencias*.

Además, hemos visto que podemos crear instrucciones más complejas a partir de otras más simples. Es decir:

- Podemos crear *expresiones más complejas* combinando entre sí expresiones más simples.
- Podemos crear *sentencias compuestas* (estructuras de control, como bloques, condicionales, bucles, etc.) combinando entre sí otras sentencias.

La propiedad que tienen los lenguajes de programación de crear elementos más complejos combinando otros más simples se denomina **composición**.

La **abstracción** y la **composición** son dos conceptos relacionados:

- *Componer* consiste en combinar elementos entre sí para formar otros más complejos.
- *Abstrer* consiste en coger un elemento (simple o complejo), darle un nombre y ocultar sus detalles internos dentro de una caja negra.

Lo interesante es que la combinación y la abstracción son dos mecanismos *recursivos*:

- Podemos crear elementos complejos a partir de otros elementos complejos.
- Podemos crear abstracciones a partir de otras abstracciones.

Además, por supuesto, podemos crear abstracciones a partir de composiciones y composiciones a partir de abstracciones.

Por ahora, esos conceptos (*composición* y *abstracción*) sólo los hemos aplicado a las **instrucciones** del programa:

- La **composición de instrucciones** da lugar a las **expresiones compuestas** y a las **sentencias compuestas** (también llamadas **estructuras de control**: *secuencia*, *selección* e *iteración*).
- La **abstracción de instrucciones** da lugar a las **abstracciones funcionales**.

Pero también se pueden aplicar a los **datos**:

- La **composición de datos** da lugar a los **datos compuestos** (también llamados **datos estructurados**) y, en consecuencia, a los **tipos de datos compuestos** (también llamados **tipos de datos estructurados**).
- La **abstracción de datos** da lugar a los **datos abstractos** y, en consecuencia, a los **tipos abstractos de datos**.

En esta unidad hablaremos de los primeros, y dejaremos los segundos para una unidad posterior.

1.2. Conceptos básicos

Un **dato estructurado** (también llamado **dato compuesto**, **colección** o **contenedor**) es un dato formado, a su vez, por otros datos llamados **componentes** o **elementos**.

Un **tipo de dato estructurado**, también llamado **tipo compuesto**, es aquel cuyos valores son datos estructurados.

Normalmente, se puede **acceder** de manera individual a los elementos que componen un dato estructurado y, a veces, también se pueden **modificar** de manera individual.

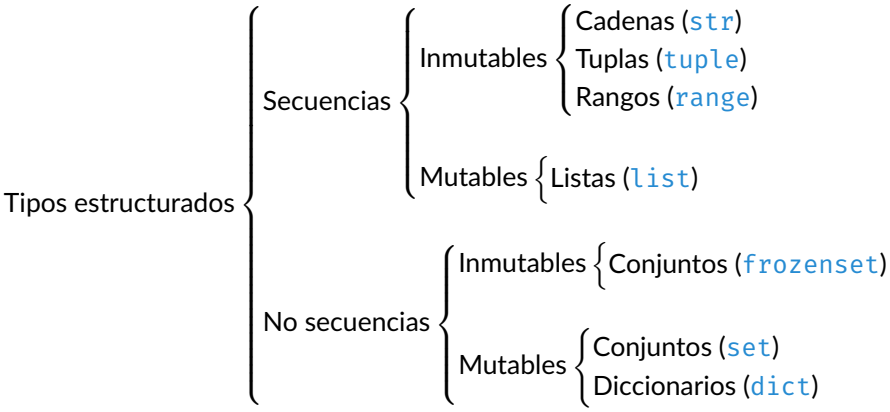
El término **estructura de datos** se suele usar como sinónimo de **tipo de dato estructurado**, aunque nosotros haremos una distinción:

- Usaremos **tipo de dato estructurado** cuando usemos un dato sin conocer sus detalles internos de implementación.
- Usaremos **estructura de datos** cuando nos interesen esos detalles internos.

1.3. Clasificación

Los **datos estructurados** se pueden clasificar en:

- **Secuenciales:** los elementos se pueden acceder directamente según la posición que ocupan dentro de la secuencia.
- **No secuenciales:** los elementos no se pueden acceder directamente según la posición que ocupan, normalmente porque esos los elementos no se encuentran en una posición concreta dentro de la secuencia.
- **Inmutables:** el dato estructurado no puede cambiar nunca su estado interno a lo largo de su vida.
- **Mutables:** el dato estructurado puede cambiar su estado interno a lo largo de su vida sin cambiar su identidad.



1.4. *Hashables*

Un dato es *hashable* si cumple las siguientes dos condiciones:

1. Tiene asociado un valor numérico llamado ***hash*** que nunca cambia durante su vida.

   Si un dato es *hashable*, se podrá obtener su *hash* llamando a la función `hash` sobre el valor. En caso contrario, la llamada generará un error `TypeError`.

2. Puede compararse con otros datos usando el operador `==`.

Si dos datos *hashables* son iguales, entonces deben tener el mismo valor de *hash*:

> Si *x* `==` *y*, entonces debe cumplirse que `hash(`*x*`) == hash(`*y*`)`.

La mayoría de los datos inmutables predefinidos en Python son *hashables*.

Los **contenedores inmutables** (como las tuplas o los `frozenset`s) sólo son *hashables* si sus elementos también lo son.

Los **contenedores mutables** (como las listas o los diccionarios) **NO** son *hashables*.

El concepto de *hashable* es importante en Python ya que existen tipos de datos estructurados que sólo admiten elementos *hashables*.

Por ejemplo, los elementos de un conjunto y las claves de un diccionario deben ser *hashables*.

El valor *hash* de un dato se calcula internamente a partir del contenido del dato usando una fórmula que no nos debe preocupar por ahora.

Ese valor se utiliza para acceder directamente al dato dentro de una colección, y por eso es un valor que no puede cambiar nunca.

Esa es la razón por la que los contenedores mutables no son *hashables*: al ser mutables, su contenido cambia y, por tanto, su valor *hash* también.

Ejemplos:

```
>>> hash('hola')
1466824599200729805
>>> hash(5)
5
>>> hash((1, 2, 3))
529344067295497451
>>> hash([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

No se debe confundir el `id` de un dato con el `hash` de un dato:

`id`	`hash`
El `id` devuelve la identidad de un dato.	El `hash` devuelve el *hash* de un dato, si es *hashable*.
Todos los datos tienen identidad.	No todos los datos son *hashables*.

`id`	`hash`
Puede haber datos iguales pero no idénticos.	Si dos datos son iguales, sus *hash* también deben serlo.
Su valor es independiente de la información concreta que contenga el dato.	Su valor se obtiene a partir de la información que contiene el dato, usando una fórmula matemática.
Por tanto, no cambia si se modifica el dato.	Por tanto, un dato mutable no puede ser *hashable*.

El *hash* de un dato es un número que representa al dato y a todo su contenido de una manera bien definida según una fórmula.

En cierto modo, ese número *resume* el contenido del dato en un simple número entero.

Gracias a ello, el intérprete puede utilizar técnicas que permiten localizar directamente a un dato dentro de una colección, de forma casi inmediata y sin importar el tamaño de la colección.

De lo contrario, el intérprete tendría que buscar secuencialmente el dato dentro de la colección desde el principio hasta el final, lo que resultaría mucho más lento y consumiría un tiempo proporcional al tamaño de la colección (cuanto más grande sea la colección, más tardará).

En definitiva, los *hash* **permiten el acceso directo a un dato** dentro de una colección.

Muy en resumen, la técnica consiste en dividir el espacio en memoria que ocupa la colección en una serie de *contenedores* llamados ***buckets***.

Cada *bucket* va numerado por un posible valor de *hash*, de forma que el *bucket* número *n* contendrá todos los elementos cuyo *hash* valga *n*.

Por tanto, el algoritmo que usa el intérprete para encontrar un elemento *hashable* dentro de una colección es:

1. Calcular el *hash* del elemento a localizar.
2. Irse directamente al *bucket* numerado con ese valor de *hash* (esta es una operación casi inmediata, con coste $O(1)$).
3. Localizar dentro del *bucket* aquel elemento que sea igual al que se está buscando, usando el `==`.

Al final, se consigue encontrar al elemento (si existe) de forma muy rápida, con un coste que es aproximadamente constante, independientemente de la cantidad de elementos que haya en la colección.

1.5. Iterables

Un **iterable** es un dato compuesto que se puede recorrer o visitar elemento a elemento, es decir, que se puede *iterar* por sus elementos uno a uno.

Como iterables tenemos:

- Todas las secuencias: listas, cadenas, tuplas y rangos
- Estructuras no secuenciales: diccionarios y conjuntos

No representa un tipo concreto, sino más bien una *familia* de tipos que comparten la misma propiedad.

Muchas funciones, como `map` y `filter`, actúan sobre iterables en general, en lugar de hacerlo sobre un tipo concreto.

La forma básica de recorrer un dato iterable es usando un **iterador**.

De hecho, un *iterable* es cualquier dato que lleva asociado, al menos, un *iterador*.

1.6. Iteradores

Un **iterador** representa un flujo de datos *perezoso* (no se entregan todos de una vez, sino de uno en uno).

Cuando se llama repetidamente a la función `next` aplicada a un iterador, se van obteniendo los sucesivos elementos del flujo.

Cuando ya no hay más elementos disponibles, se levanta una excepción de tipo `StopIteration`.

Eso indica que el iterador se ha agotado, por lo que si se sigue llamando a la función `next` se seguirá levantando esa excepción.

Se puede obtener un iterador a partir de cualquier dato iterable aplicando la función `iter` al iterable.

Si se le pasa un dato no iterable, levanta una excepción `TypeError`.

Ejemplo:

```
>>> lista = [1, 2, 3, 4]
>>> it = iter(lista)
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
4
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

También se suele decir que **los iteradores son iterables perezosos de un solo uso**:

- Son **perezosos** porque calculan sus elementos a medida que los vas recorriendo.
- Son **de un solo uso** porque, una vez que se ha consumido un elemento, ya no vuelve a aparecer.

Se dice que un iterador está **agotado** si se ha consumido completamente, es decir, si se han consumido todos sus elementos.

Funciones como `map` y `filter` devuelven iteradores porque, al ser perezosos, son más eficiente en memoria que devolver toda una lista o tupla.

Por ejemplo: ¿qué ocurre si sólo necesitamos los primeros elementos del resultado de un `map`?

Los iteradores se pueden convertir en listas o tuplas usando las funciones `list` y `tuple`:

```
>>> l = [1, 2, 3]
>>> iterador = iter(l)
>>> t = tuple(iterador)
>>> t
(1, 2, 3)
```

Las **expresiones generadoras**, ya conocidas por nosotros, también son expresiones que **devuelven un iterador**:

⟨*expr_gen*⟩ ::= **(**⟨*expresión*⟩ (**for** ⟨*identificador*⟩ **in** ⟨*secuencia*⟩ [**if** ⟨*condición*⟩])+**)**

Ejemplo:

```
>>> cuadrados = (x ** 2 for x in range(1, 10))
>>> cuadrados
<generator object <genexpr> at 0x7f6a0fc7db48>
>>> next(cuadrados)
1
>>> next(cuadrados)
4
>>> next(cuadrados)
9
```

Los iteradores también son iterables que actúan como sus propios iteradores.

Por tanto, cuando llamamos a `iter` pasándole un iterador, se devuelve el mismo iterador:

```
>>> lista = [1, 2, 3, 4]
>>> it = iter(lista)
>>> it
<list_iterator object at 0x7f3c49aa9080>
>>> it2 = iter(it)
>>> it2
<list_iterator object at 0x7f3c49aa9080>
```

Por tanto, también podemos usar un iterador en cualquier sitio donde se espere un iterable.

1.6.1. El bucle `for`

Probablemente, la mejor forma de recorrer los elementos que devuelve un iterador es mediante una **estructura de control** llamada **bucle `for`**.

Su sintaxis es:

for ⟨*variable*⟩(**,** ⟨*variable*⟩)* **in** ⟨*iterable*⟩**:**
    ⟨*sentencia*⟩

que no es más que azúcar sintáctico para el siguiente código equivalente:

```
iterador = iter(⟨iterable⟩)
fin = False
```

```
while not fin:
    try:
        ⟨variable⟩(, ⟨variable⟩)* = next(iterador)
    except StopIteration:
        fin = True
    else:
        ⟨sentencia⟩
```

Ejemplos:

```
for i in range(0, 4):
    print(i)
```

devuelve:

```
0
1
2
3
```

```
for x in ['hola', 23.5, 10, [1, 2]]:
    print(x * 2)
```

devuelve:

```
'holahola'
47.0
20
[1, 2, 1, 2]
```

Al recorrer la lista, la variable va almacenando en cada paso el valor del elemento que en ese momento se está visitando.

Si necesitáramos recuperar también el índice del elemento, podemos usar la función `enumerate`.

Esta función devuelve un iterador que va generando tuplas que contienen, además del elemento, un valor numérico que representa un contador.

Las tuplas que devuelve el iterador llevan el contador en la primera posición y el elemento de la lista en la segunda posición.

Ese contador, por defecto, empieza desde `0` y se va incrementando de uno en uno, por lo que coincide con el índice del elemento en la lista:

```
>>> for i, e in enumerate(['a', 'b', 'c']):
...     print('El elemento en la posición ' + str(i) + ' es ' + str(e))
...
El elemento en la posición 0 es a
El elemento en la posición 1 es b
El elemento en la posición 2 es c
```

Existen iterables e iteradores incluso donde uno menos se lo podría esperar.

Por ejemplo, **los archivos abiertos también son iterables**, ya que se pueden recorrer línea a línea usando un iterador:

```
with open('archivo.txt') as f:
    for linea in f:
        print(linea)
```

Esta forma de recorrer los archivos, además de resultar simple y elegante, también resulta muy eficiente, ya que se va recuperando cada línea de una en una en lugar de todas a la vez.

1.6.2. El módulo `itertools`

El módulo `itertools` contiene una variedad de iteradores de uso frecuente así como funciones que combinan varios iteradores.

`itertools.count(`[⟨*inicio*⟩[`,` ⟨*paso*⟩]]`)` devuelve un flujo infinito de valores separados uniformemente. Se puede indicar opcionalmente un valor de comienzo (que por defecto es `0`) y el intervalo entre números (que por defecto es `1`):

- `itertools.count()` ⇒ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
- `itertools.count(10)` ⇒ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
- `itertools.count(10, 5)` ⇒ 10, 15, 20, 25, 30, 35, 40, 45, ...

`itertools.cycle(`⟨*iterador*⟩`)` devuelve un nuevo iterador que va generando sus elementos del primero al último, repitiéndolos indefinidamente:

- `itertools.cycle([1, 2, 3, 4])` ⇒ 1, 2, 3, 4, 1, 2, 3, 4, ...

`itertools.repeat(`⟨*elem*⟩[`,` ⟨*n*⟩]`)` devuelve ⟨*n*⟩ veces el elemento ⟨*elem*⟩, o lo devuelve indefinidamente si no se indica ⟨*n*⟩:

- `itertools.repeat('abc')` ⇒ abc, abc, abc, abc, abc, abc, abc, ...
- `itertools.repeat('abc', 5)` ⇒ abc, abc, abc, abc, abc

2. Secuencias

2.1. Concepto de secuencia

Una **secuencia** s es un dato estructurado que cumple lo siguiente:

- Se le puede calcular su longitud (la cantidad de elementos que contiene) mediante la función `len`.
- Cada elemento que contiene lleva asociado un número entero llamado **índice**, comprendido entre `0` y `len(`*s*`) - 1`.
- Permite el acceso eficiente a cada uno de sus elementos mediante indexación *s*`[`*i*`]`, siendo *i* el índice del elemento.

Las secuencias se dividen en:

- **Inmutables**: cadenas (`str`), tuplas (`tuple`) y rangos (`range`).
- **Mutables**: listas (`list`)

2.2. Operaciones comunes

Todas las secuencias (ya sean cadenas, listas, tuplas o rangos) comparten un conjunto de **operaciones comunes**.

Los rangos son una excepción, ya que sus elementos se crean a partir de una fórmula y, por eso, no admiten ni la concatenación ni la repetición.

La siguiente tabla recoge las operaciones comunes sobre secuencias, ordenadas por prioridad ascendente. *s* y *t* son secuencias del mismo tipo, *n*, *i*, *j* y *k* son enteros y *x* es un dato cualquiera que cumple con las restricciones que impone *s*.

Operación	Resultado
`x in s`	`True` si algún elemento de *s* es igual a *x*
`x not in s`	`False` si algún elemento de *s* es igual a *x*
`s + t`	La concatenación de *s* y *t* (no va con rangos)
`s * n` `n * s`	(*Repetición*) Equivale a concatenar *s* consigo misma *n* veces (no va con rangos)
`s[i]`	El *i*-ésimo elemento de *s*, empezando por 0
`s[i:j]`	Rodaja de *s* desde *i* hasta *j*
`s[i:j:k]`	Rodaja de *s* desde *i* hasta *j* con paso *k*
`len(s)`	Longitud de *s*
`min(s)`	El elemento más pequeño de *s*
`max(s)`	El elemento más grande de *s*
`s.index(x[, i[, j]])`	El índice de la primera aparición de *x* en *s* (desde el índice *i* inclusive y antes del *j*)
`s.count(x)`	Número total de apariciones de *x* en *s*

Además de estas operaciones, las secuencias admiten **comparaciones** con los operadores `==`, `!=`, `<`, `<=`, `>` y `>=`.

Dos secuencias *s* y *t* son iguales (`s == t`) si:

- Son del mismo tipo (`type(s) == type(t)`).
- Tienen la misma longitud (`len(s) == len(t)`).
- Contienen los mismos elementos en el mismo orden (`s[0] == t[0]`, `s[1] == t[1]`, etcétera).

Por supuesto, las dos secuencias son distintas (`s != t`) si no son iguales.

Se pueden comparar dos secuencias con los operadores `<`, `<=`, `>` y `>=` para comprobar si una es menor (o igual) o mayor (o igual) que la otra si:

- Son del mismo tipo (si no son del mismo tipo, lanza una excepción).

- No son rangos.

Las comparaciones `<`, `<=`, `>` y `>=` se hacen lexicográficamente elemento a elemento, como en un diccionario.

Por ejemplo, `'adios' < 'hola'` porque `adios` aparece antes que `hola` en el diccionario.

Con el resto de las secuencias se actúa igual que con las cadenas.

Dadas dos secuencias s y t, para ver si $s < t$ se procede así:

- Se empieza comparando el primer elemento de s con el primero de t.
- Si son iguales, se pasa al siguiente hasta encontrar algún elemento de s que sea distinto a su correspondiente de t.
- Si llegamos al final de s sin haber encontrado ningún elemento distinto a su correspondiente en t, es porque $s == t$.
- En cuanto se encuentre un elemento de s que no es igual a su correspondiente de s, se comparan esos elementos y se devuelve el resultado de esa comparación.

Los rangos no se pueden comparar con `<`, `<=`, `>` o `>=`.

Ejemplos:

```
>>> (1, 2, 3) == (1, 2, 3)
True
>>> (1, 2, 3) != (1, 2, 3)
False
>>> (1, 2, 3) == (3, 2, 1)
False
>>> (1, 2, 3) < (3, 2, 1)
True
>>> (1, 2, 3) < (1, 2, 4)
True
>>> (1, 2, 3) < (1, 2, 4, 5)
True
>>> 'hola' < 'adios'
False
>>> range(0, 3) < range(3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'range' and 'range'
```

2.3. Inmutables

2.3.1. Cadenas (`str`)

Las **cadenas** son secuencias inmutables y *hashables* de caracteres.

No olvidemos que en Python no existe el tipo *carácter*. En Python, un carácter es una cadena de longitud 1.

Las cadenas literales se pueden crear:

- Con comillas simples (`'`) o dobles (`"`):

```
>>> 'hola'
'hola'
>>> "hola"
'hola'
```

- Con triples comillas (`'''` o `"""`):

```
>>> """hola
... qué tal"""
'hola\nqué tal'
```

Las cadenas implementan todas las operaciones de las secuencias, además de los métodos que se pueden consultar en https://docs.python.org/3/library/stdtypes.html#string-methods

2.3.1.1. Formateado de cadenas

Una **cadena formateada** (también llamada ***f-string***) es una cadena literal que lleva un prefijo `f` o `F`.

Estas cadenas contienen **campos de sustitución**, que son expresiones encerradas entre llaves.

En realidad, las cadenas formateadas son expresiones evaluadas en tiempo de ejecución.

Sintaxis:

⟨*f_string*⟩ ::= (⟨*carácter_literal*⟩ | **{{** | **}}** | ⟨*sustitución*⟩)*
⟨*sustitución*⟩ ::= **{**⟨*expresión*⟩ [**!**⟨*conversión*⟩] [**:**⟨*especif*⟩]**}**
⟨*conversión*⟩ ::= **s** | **r** | **a**
⟨*especif*⟩ ::= (⟨*carácter_literal*⟩ | **NULL** | ⟨*sustitución*⟩)*
⟨*carácter_literal*⟩ ::= ⟨*cualquier carácter Unicode excepto* **{**, **}** *o* **NULL**⟩

Las partes de la cadena que van fuera de las llaves se tratan literalmente, excepto las dobles llaves `{{` y `}}`, que son sustituidas por una sola llave.

Una `{` marca el comienzo de un **campo de sustitución** (⟨*sustitución*⟩), que empieza con una expresión.

Tras la expresión puede venir un **conversión** (⟨*conversión*⟩), introducida por una exclamación `!`.

También puede añadirse un **especificador de formato** (⟨*especif*⟩) después de dos puntos `:`.

El campo de sustitución termina con una `}`.

Las expresiones en un literal de cadena formateada son tratadas como cualquier otra expresión Python encerrada entre paréntesis, con algunas excepciones:

- No se permiten expresiones vacías.
- Las expresiones lambda deben ir entre paréntesis.

Los campos de sustitución pueden contener saltos de línea pero no comentarios.

Si se indica una conversión, el resultado de evaluar la expresión se convierte antes de aplicar el formateado.

La conversión `!s` llama a la función `str` sobre el resultado, `!r` llama a `repr` y `!a` llama a `ascii`.

A continuación, el resultado es formateado usando la función `format`.

Finalmente, el resultado del formateado es incluido en el valor final de la cadena completa.

La sintaxis general de un especificador de formato es:

```
⟨especif⟩ ::= [[⟨relleno⟩]⟨alig⟩][⟨signo⟩][#][0][⟨ancho⟩][⟨grupos⟩][.⟨precision⟩][⟨tipo⟩]
⟨relleno⟩ ::= ⟨cualquier carácter⟩
⟨alig⟩ ::= < | > | = | ^
⟨signo⟩ ::= + | - | ⟨espacio⟩
⟨ancho⟩ ::= ⟨dígito⟩+
⟨grupos⟩ ::= _ | ,
⟨precision⟩ ::= ⟨dígito⟩+
⟨tipo⟩ ::= b | c | d | e | E | f | F | g | G | n | o | s | x | X | %
```

Los especificadores de formato de nivel superior pueden incluir campos de sustitución anidados.

Estos campos anidados pueden incluir, a su vez, sus propios campos de conversión y sus propios especificadores de formato, pero no pueden incluir más campos de sustitución anidados.

Para más información, consultar https://docs.python.org/3.7/library/string.html#format-specification-mini-language

Algunos ejemplos de cadenas formateadas:

```
>>> nombre = "Fred"
>>> f"Dice que su nombre es {nombre!r}."
"Dice que su nombre es 'Fred'."
>>> f"Dice que su nombre es {repr(nombre)}."  # repr es equivalente a !r
"Dice que su nombre es 'Fred'."
>>> ancho = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{ancho}.{precision}}"  # campos anidados
'result:      12.35'
>>> import datetime
>>> hoy = datetime.datetime(year=2017, month=1, day=27)
>>> f"{hoy:%B %d, %Y}"  # usando especificador de formato de fecha
'January 27, 2017'
>>> numero = 1024
>>> f"{numero:#0x}"  # usando especificador de formato de enteros
'0x400'
```

2.3.1.2. Expresiones regulares

Las **expresiones regulares** (también llamados *regex*) constituyen un pequeño lenguaje muy especializado incrustado dentro de Python y disponible a través del módulo `re`.

Usando este pequeño lenguaje es posible especificar **reglas sintácticas** de una forma distinta pero parecida a las *gramáticas EBNF* (aunque con menos poder expresivo).

Esas reglas sintácticas se pueden usar luego para **comprobar si una cadena *se ajusta* a un patrón**.

Este patrón puede ser frases en español, o direcciones de correo electrónico o cualquier otra cosa.

A continuación, se pueden hacer preguntas del tipo: «*¿Esta cadena se ajusta al patrón?*» o «*¿Hay algo que se ajuste al patrón en alguna parte de esta cadena?*».

También se pueden usar las *regexes* para **modificar** una cadena o **dividirla en partes** según el patrón indicado.

El lenguaje de las expresiones regulares es relativamente pequeño y restringido, por lo que no es posible usarlo para realizar cualquier tipo de procesamiento de cadenas.

Además, hay procesamientos que se pueden realizar con *regexes* pero las expresiones que resultan se vuelven muy complicadas.

En estos casos, es mejor escribir directamente código Python ya que, aunque el código resultante pueda resultar más lento, probablemente resulte más fácil de leer.

Para más información sobre cómo crear y usar expresiones regulares, consultar:

- Tutorial de introducción en https://docs.python.org/3/howto/regex.html
- Documentación del módulo `re` en https://docs.python.org/3/library/re.html

2.3.2. Tuplas

Las **tuplas** (`tuple`) son secuencias inmutables, usadas frecuentemente para representar colecciones de datos heterogéneos (es decir, de tipos distintos).

También se usan en aquellos casos en los que se necesita una secuencia inmutable de datos homogéneos (por ejemplo, para almacenar datos en un conjunto o un diccionario).

Las tuplas se pueden crear así:

- Si es una tupla vacía, con paréntesis vacíos: `()`
- Si sólo tiene un elemento, se pone una coma detrás:

  a`,`

  `(`*a*`,)`
- Si tiene más de un elemento, se separan con comas:

  a`,` *b*`,` *c*

  `(`*a*`,` *b*`,` *c*`)`
- Usando la función `tuple(`⟨*iterable*⟩`)`.

Observar que lo que construye la tupla es realmente la coma, no los paréntesis.

Los paréntesis son opcionales, excepto en dos casos:

- La tupla vacía: `()`
- Cuando son necesarios para evitar ambigüedad.

  Por ejemplo, `f(`*a*`,` *b*`,` *c*`)` es una llamada a una función con tres argumentos, mientras que `f((`*a*`,` *b*`,` *c*`))` es una llamada a una función con un único argumento que es una tupla de tres elementos.

Las tuplas implementan todas las operaciones comunes de las secuencias.

En general, las tuplas se pueden considerar como la versión inmutable de las listas.

Además, las tuplas son *hashables* si sus elementos también lo son.

2.3.3. Rangos

Los **rangos** (`range`) representan secuencias inmutables y *hashables* de números enteros y se usan frecuentemente para hacer bucles que se repitan un determinado número de veces.

Los rangos se crean con la función `range`:

```
range([start: int,] stop: int [, step: int]) -> range
```

Cuando se omite *start*, se entiende que es `0`.

Cuando se omite *step*, se entiende que es `1`.

El valor de *stop* no se alcanza nunca.

Cuando *start* y *stop* son iguales, representa el *rango vacío*.

step debe ser siempre distinto de cero.

Cuando *start* es mayor que *stop*, el valor de *step* debería ser negativo. En caso contrario, también representaría el rango vacío.

El **contenido** de un rango r vendrá determinado por la fórmula:

$$r[i] = start + step \cdot i$$

donde $i \geq 0$. Además:

- Si $step > 0$, se impone también la restricción $r[i] < stop$.
- Si $step < 0$, se impone también la restricción $r[i] > stop$.

Un rango es **vacío** cuando $r[0]$ no satisface las restricciones anteriores.

Los rangos admiten **índices negativos**, pero se interpretan como si se indexara desde el final de la secuencia usando índices positivos.

Los rangos implementan **todas las operaciones de las secuencias, *excepto* la concatenación y la repetición**.

Esto es debido a que los rangos sólo pueden representar secuencias que siguen un patrón muy estricto, y las repeticiones y las concatenaciones a menudo violan ese patrón.

Los rangos son perezosos y además ocupan mucha menos memoria que las listas o las tuplas (sólo hay que almacenar *start*, *stop* y *step*).

La **forma normal** de un rango es una expresión en la que se llama a la función `range` con los argumentos necesarios para construir el rango:

```
>>> range(10)
range(0, 10)
>>> range(0, 10)
range(0, 10)
>>> range(0, 10, 1)
range(0, 10)
```

```
>>> range(0, 30, 5)
range(0, 30, 5)
>>> range(0, -10, -1)
range(0, -10, -1)
>>> range(4, -5, -2)
range(4, -5, -2)
```

Para ver con claridad todos los elementos de un rango, podemos convertirlo en una tupla o una lista. Por ejemplo:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Dos rangos son considerados **iguales** si representan la misma secuencia de valores, sin importar si tienen distintos valores de *start*, *stop* o *step*.

Por ejemplo:

```
>>> range(20) == range(0, 20)
True
>>> range(0, 20) == range(0, 20, 2)
False
>>> range(0, 3, 2) == range(0, 4, 2)
True
>>> range(0) == range(2, 1, 3)
True
```

El **rango vacío** es un valor que no tiene expresión canónica, ya que cualquiera de las siguientes expresiones representan al rango vacío tan bien como cualquier otra:

- `range(0)`.
- `range(a, a)`, donde a es cualquier entero.
- `range(a, b, c)`, donde $a \geq b$ y $c > 0$.
- `range(a, b, c)`, donde $a \leq b$ y $c < 0$.

```
>>> range(3, 3) == range(4, 4)
True
>>> range(4, 3) == range(3, 4, -1)
True
```

2.4. Mutables

2.4.1. Listas

Las **listas** son secuencias *mutables*, usadas frecuentemente para representar colecciones de elementos heterogéneos.

Al ser mutables, las listas **no** son *hashables*.

Se pueden construir de varias maneras:

- Usando corchetes vacíos para representar la lista vacía: `[]`.
- Usando corchetes y separando los elementos con comas:

  `[`*a*`]`

  `[`*a*`, `*b*`, `*c*`]`
- Usando la función `list` con las sintaxis `list()` o `list(`⟨*iterable*⟩`)`.

La función `list` construye una lista cuyos elementos son los mismos (y están en el mismo orden) que los elementos de ⟨*iterable*⟩.

⟨*iterable*⟩ puede ser:

- una secuencia,
- un contenedor sobre el que se pueda iterar, o
- un iterador.

Si se llama sin argumentos, devuelve una lista vacía.

Por ejemplo:

```
>>> list('hola')
['h', 'o', 'l', 'a']
>>> list((1, 2, 3))
[1, 2, 3]
```

2.4.1.1. Listas por comprensión

También se pueden crear **listas por comprensión** usando la misma sintaxis de las **expresiones generadoras** pero encerrando la expresión entre corchetes en lugar de entre paréntesis:

```
>>> [x ** 2 for x in [1, 2, 3]]
[1, 4, 9]
```

Como se ve, el resultado es directamente una lista, no un iterador.

Por tanto, a diferencia de lo que pasa con las expresiones generadoras, **el resultado de una lista por comprensión no es perezoso**, cosa que habrá que tener en cuenta para evitar consumir más memoria de la necesaria o generar elementos que al final no sean necesarios.

Por ejemplo, la siguiente expresión generadora:

```
res_gen = (x ** 2 for x in range(0, 10000000000000))
```

es mucho más eficiente en tiempo y espacio que la lista por comprensión:

```
res_list = [x ** 2 for x in range(0, 10000000000000)]
```

ya que la expresión generadora devuelve un **iterador** que irá generando los valores de uno en uno a medida que los vayamos recorriendo con `next(res_gen)`.

En cambio, la lista por comprensión genera todos los valores de la lista a la vez y los almacena todos juntos en la memoria.

A cambio, la ventaja de tener una lista frente a tener un iterador es que podemos acceder directamente a cualquier elemento de la lista mediante la indexación.

Las listas por comprensión, al igual que las expresiones generadoras, **determinan su propio *ámbito***.

Ese ámbito abarca toda la lista por comprensión, de principio a fin.

Los identificadores que aparecen en la cláusula **`for`** se consideran *variables ligadas* en la lista por comprensión.

Esos identificadores se van ligando, uno a uno, a cada elemento de la secuencia indicada en la cláusula **`in`**.

Como son variables ligadas, cumplen estas dos propiedades:

- Se pueden renombrar (siempre de forma consistente) sin que la lista por comprensión cambie su significado.

  Por ejemplo, las dos listas por comprensión siguientes son equivalentes, puesto que producen el mismo resultado:

  ```
  [x for x in (1, 2, 3)]
  ```

  ```
  [y for y in (1, 2, 3)]
  ```

- No se pueden usar fuera de la lista por comprensión, ya que estarían fuera de su ámbito y no serían visibles.

  Por ejemplo, lo siguiente daría un error de nombre:

  ```
  >>> e = [x for x in (1, 2, 3)]
  >>> x       # Intento acceder a la 'x' de la lista por comprensión
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  NameError: name 'x' is not defined
  ```

2.4.2. Operaciones mutadoras

En la siguiente tabla, *s* es una instancia de un tipo de secuencia mutable (por ejemplo, una lista), *t* es cualquier dato iterable y *x* es un dato cualquiera que cumple con las restricciones que impone *s*:

Operación	Resultado
`s[i] = x`	El elemento *i*-ésimo de *s* se sustituye por *x*
`s[i:j] = t`	La rodaja de *s* desde *i* hasta *j* se sustituye por *t*
`s[i:j:k] = t`	Los elementos de `s[i:j:k]` se sustituyen por *t*
`del s[i]`	Elimina el elemento *i*-ésimo de *s*
`del s[i:j]`	Elimina los elementos de `s[i:j]` Equivale a hacer `s[i:j] = []`
`del s[i:j:k]`	Elimina los elementos de `s[i:j:k]`

Operación	Resultado
`s.append(x)`	Añade *x* al final de *s*; es igual que `s[len(s):len(s)] = [x]`
`s.clear()`	Elimina todos los elementos de *s*; es igual que `del s[:]`
`s.copy()`	Crea una copia *superficial* de *s*; es igual que `s[:]`
`s.extend(t)` `s += t`	Extiende *s* con el contenido de *t*; es igual que `s[len(s):len(s)] = t`
`s *= n`	Modifica *s* repitiendo su contenido *n* veces
`s.insert(i, x)`	Inserta *x* en *s* en el índice *i*; es igual que `s[i:i] = [x]`
`s.pop([i])`	Extrae el elemento *i* de *s* y lo devuelve (por defecto, *i* vale `-1`)
`s.remove(x)`	Quita el primer elemento de *s* que sea igual a *x*
`s.reverse()`	Invierte los elementos de *s*
`s.sort()`	Ordena los elementos de *s*

La **copia superficial** (a diferencia de la ***copia profunda***) significa que sólo se copia el objeto sobre el que se aplica la copia, **no sus elementos**.

Por tanto, al crear la copia superficial, se crea sólo un nuevo objeto, donde se copiarán las referencias de los elementos del objeto original.

Esto influye, sobre todo, cuando los elementos de una colección mutable también son objetos mutables.

Por ejemplo, si tenemos listas dentro de otra lista, y copiamos ésta última con `.copy()`, la nueva lista compartirá elementos con la lista original:

```
>>> x = [[1, 2], [3, 4]]   # los elementos de «x» también son listas
>>> y = x.copy()           # «y» es una copia de «x»
>>> x is y
False                      # no son la misma lista
>>> x[0] is y[0]
True                       # sus elementos no se han copiado,
>>> x[0].append(9)         # sino que están compartidos por «x» e «y»
>>> x
[[1, 2, 99], [3, 4]]
>>> y
[[1, 2, 99], [3, 4]]
```

El método `sort` permite ordenar los elementos de la secuencia de forma ascendente o descendente:

```
>>> x = [3, 6, 2, 9, 1, 4]
>>> x.sort()
>>> x
[1, 2, 3, 4, 6, 9]
>>> x.sort(reverse=True)
>>> x
[9, 6, 4, 3, 2, 1]
```

3. Estructuras no secuenciales

3.1. Conjuntos (`set` y `frozenset`)

Un conjunto es una colección **no ordenada** de elementos *hashables*.

Se usan frecuentemente para comprobar si un elemento pertenece a un grupo, para eliminar duplicados en una secuencia y para realizar operaciones matemáticas como la *unión*, la *intersección* y la *diferencia simétrica*.

Como cualquier otra colección, los conjuntos permiten el uso de:

- `x in c`
- `len(c)`
- `for x in c`

Como son colecciones no ordenadas, los conjuntos **no almacenan la posición** de los elementos o el **orden** en el que se insertaron.

Por tanto, tampoco admiten la indexación, las rodajas ni cualquier otro comportamiento propio de las secuencias.

Cuando decimos que **un conjunto no está ordenado**, queremos decir que los elementos que contiene no se encuentran situados en una posición concreta.

- Es lo contrario de lo que ocurre con las sencuencias, donde cada elemento se encuentra en una posición indicada por su *índice* y podemos acceder a él usando la indexación.

Además, en un conjunto **no puede haber elementos repetidos** (un elemento concreto sólo puede estar *una vez* dentro de un conjunto, es decir, o está una vez o no está).

En resumen:

> **En un conjunto:**
>
> Un elemento concreto, o está una vez, o no está.
>
> Si está, no podemos saber en qué posición (no tiene sentido preguntárselo).

Existen dos tipos predefinidos de conjuntos: `set` y `frozenset`.

El tipo `set` es **mutable**, es decir, que su contenido puede cambiar usando métodos como `add` y `remove`.

- Como es mutable, **no es *hashable*** y, por tanto, no puede usarse como clave de un diccionario o como elemento de otro conjunto.

El tipo `frozenset` es **inmutable** y *hashable*. Por tanto, su contenido no se puede cambiar una vez creado y puede usarse como clave de un diccionario o como elemento de otro conjunto.

Los dos tipos de conjuntos se crean con las funciones `set(`[⟨*iterable*⟩]`)` y `frozenset(`[⟨*iterable*⟩]`)`.

- Si se llaman *sin argumentos*, devuelven un conjunto **vacío**:
  - `set()` devuelve un conjunto vacío de tipo `set`.
  - `frozenset()` devuelve un conjunto vacío de tipo `frozenset`.

```
>>> set()
set()
>>> frozenset()
frozenset()
```

  Como se ve, esas son, precisamente, las **formas normales** de un conjunto vacío de tipo `set` y `frozenset`.

- Si se les pasa un *iterable* (como por ejemplo, una lista), devuelve un conjunto formado por los elementos del iterable:

```
>>> set([4, 3, 2, 2, 4])
{2, 3, 4}
>>> frozenset([4, 3, 2, 2, 4])
frozenset({2, 3, 4})
```

Además, existe una *sintaxis especial* para escribir **literales de conjuntos no vacíos de tipo `set`**, que consiste en encerrar sus elementos entre llaves y separados por comas: `{'pepe', 'juan'}`.

```
>>> x = {'pepe', 'juan'}  # un literal de tipo set, como set(['pepe', 'juan'])
>>> x
{'pepe', 'juan'}
>>> type(x)
<class 'set'>
```

Esa es, precisamente, la **forma normal** de un conjunto no vacío y, por tanto, la que se usa cuando se visualiza desde el intérprete o se imprime con `print`.

Por tanto, para crear conjuntos congelados usando `frozenset` podemos usar esa sintaxis en lugar de usar listas como hicimos antes:

```
>>> frozenset({4, 3, 2, 2, 4})
frozenset({2, 3, 4})
```

También podría usarse con la función `set`, pero sería innecesario y no tendría sentido, ya que devolvería el mismo conjunto:

```
>>> set({4, 3, 2, 2, 4})  # equivale a poner simplemente {4, 3, 2, 2, 4}
{2, 3, 4}
```

3.1.0.1. Conjuntos por comprensión

También se pueden crear **conjuntos por comprensión** usando la misma sintaxis de las **expresiones generadoras** y las **listas por comprensión**, pero esta vez encerrando la expresión entre llaves:

```
>>> {x ** 2 for x in [1, 2, 3]}
{1, 4, 9}
```

El resultado es directamente un valor de tipo `set`, no un iterador.

Los conjuntos por comprensión, al igual que las expresiones generadoras y las listas por comprensión, **determinan su propio *ámbito***.

Ese ámbito abarca todo el conjunto por comprensión, de principio a fin.

Los identificadores que aparecen en la cláusula **`for`** se consideran *variables ligadas* en el conjunto por comprensión.

Esos identificadores se van ligando, uno a uno, a cada elemento de la secuencia indicada en la cláusula **`in`**.

Como son variables ligadas, cumplen estas dos propiedades:

- Se pueden renombrar (siempre de forma consistente) sin que el conjunto por comprensión cambie su significado.

  Por ejemplo, los dos conjuntos por comprensión siguientes son equivalentes, puesto que producen el mismo resultado:

  ```
  {x for x in (1, 2, 3)}
  ```

  ```
  {y for y in (1, 2, 3)}
  ```

- No se pueden usar fuera del conjunto por comprensión, ya que estarían fuera de su ámbito y no serían visibles.

  Por ejemplo, lo siguiente daría un error de nombre:

```
>>> e = {x for x in (1, 2, 3)}
>>> x       # Intento acceder a la 'x' del conjunto por comprensión
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

3.1.1. Operaciones

s y *o* son conjuntos, y *x* es un valor cualquiera:

Operación	Resultado
`len(`*s*`)`	Número de elementos de *s* (su cardinalidad)
x `in` *s*	`True` si *x* pertenece a *s*
x `not in` *s*	`True` si *x* no pertenece a *s*
s`.isdisjoint(`*o*`)`	`True` si *s* no tiene ningún elemento en común con *o*
s`.issubset(`*o*`)` *s* `<=` *o*	`True` si *s* es un subconjunto de *o*
s `<` *o*	`True` si *s* es un subconjunto propio de *o*
s`.issuperset(`*o*`)` *s* `>=` *o*	`True` si *s* es un superconjunto de *o*
s `>` *o*	`True` si *s* es un superconjunto propio de *o*

Operación	Resultado
s`.union(`*o*`)` *s* `|` *o*	Unión de *s* y *o* ($s \cup o$)
s`.intersection(`*o*`)` *s* `&` *o*	Intersección de *s* y *o* ($s \cap o$)
s`.difference(`*o*`)` *s* `-` *o*	Diferencia de *s* y *o* ($s \setminus o$)
s`.symmetric_difference(`*o*`)` *s* `^` *o*	Diferencia simétrica de *s* y *o* ($s \triangle o$)
s`.copy()`	Devuelve una copia superficial de *s*

Tanto `set` como `frozenset` admiten **comparaciones entre conjuntos**.

Suponiendo que `a` y `b` son conjuntos:

- `a == b` si y sólo si cada elemento de `a` pertenece también a `b`, y viceversa; es decir, si cada uno es un subconjunto del otro.

- `a <= b` si y sólo si `a` es un *subconjunto* de `b` (es decir, si cada elemento de `a` está también en `b`).
- `a < b` si y sólo si `a` es un *subconjunto propio* de `b` (es decir, si `a` es un subconjunto de `b`, pero no es igual a `b`).
- `a >= b` si y sólo si `a` es un *superconjunto* de `b` (es decir, si cada elemento de `b` está también en `a`).
- `a > b` si y sólo si `a` es un *superconjunto propio* de `b` (es decir, si `a` es un superconjunto de `b`, pero no es igual a `b`).

3.1.2. Operaciones sobre conjuntos mutables

Estas tablas sólo se aplica a conjuntos mutables (o sea, al tipo `set` y no al `frozenset`):

Operación	Resultado
`s.update(o)` `s |= o`	Actualiza *s* añadiendo los elementos de *o*
`s.intersection_update(o)` `s &= o`	Actualiza *s* manteniendo sólo los elementos que están en *s* y *o*
`s.difference_update(o)` `s -= o`	Actualiza *s* eliminando los elementos que están en *o*
`s.symmetric_difference_update(o)` `s ^= o`	Actualiza *s* manteniendo sólo los elementos que están en *s* y *o* pero no en ambos

Operación	Resultado
`s.add(x)`	Añade *x* a *s*
`s.remove(x)`	Elimina *x* de *s* (produce `KeyError` si *x* no está en *s*)
`s.discard(x)`	Elimina *x* de *s* si está en *s*
`s.pop()`	Elimina y devuelve un valor cualquiera de *s* (produce `KeyError` si *s* está vacío)
`s.clear()`	Elimina todos los elementos de *s*

3.2. Diccionarios (dict)

Un **diccionario** es una colección que almacena *correspondencias* (o *asociaciones*) entre valores.

Por tanto, **los elementos de un diccionario son parejas de valores llamados *clave* y *valor***, y lo que hace el diccionario es almacenar las *claves* y el *valor* que le corresponde a cada clave.

Una restricción importante es que, en un diccionario dado, **cada clave sólo puede asociarse con un único valor**.

Por tanto, **en un diccionario no puede haber claves repetidas**, es decir, que no puede haber dos elementos distintos con la misma clave.

En la práctica, eso nos sirve para identificar cada elemento del diccionario por su clave.

Además, los elementos de un diccionario son datos mutables y, por tanto, los diccionarios también son **mutables**.

Los diccionarios se pueden crear:

- Con una pareja de llaves `{}`, que representan el **diccionario vacío**.
- Encerrando entre llaves una lista de parejas ⟨*clave*⟩`:`⟨*valor*⟩ separadas por comas: `{'juan': 4098, 'pepe': 4127`

  Esa es precisamente la **forma normal** de un diccionario y, por tanto, la que se usa cuando se visualiza desde el intérprete o se imprime con `print`.
- Usando la función `dict`.

Por ejemplo:

```
>>> v1 = {}                                          # diccionario vacío
>>> v2 = dict()                                      # también diccionario vacío
>>> v1 == v2
True
>>> a = {'uno': 1, 'dos': 2, 'tres': 3}              # literal
>>> b = dict(uno=1, dos=2, tres=3)                   # argumentos con nombre
>>> c = dict([('dos', 2), ('uno', 1), ('tres', 3)])  # lista de tuplas
>>> d = dict({'tres': 3, 'uno': 1, 'dos': 2})        # innecesario
>>> e = dict(zip(['uno', 'dos', 'tres'], [1, 2, 3])) # con dos iterables
>>> a == b and b == c and c == d and d == e          # todos son iguales
True
```

Si se intenta crear un diccionario con claves repetidas, sólo se almacenará uno de los elementos que tengan la misma clave (los demás se ignoran):

```
>>> a = {'perro': 'dog', 'gato': 'cat', 'perro': 'doggy'}
>>> a
{'perro': 'doggy', 'gato': 'cat'}
```

Como se ve, la clave `'perro'` está repetida y, por tanto, sólo se almacena uno de los dos elementos con clave repetida, que siempre es el último que aparece en el diccionario. En este caso, se almacena el elemento `'perro': 'doggy'` y se ignora el `'perro': 'dog'`.

Las claves de un diccionario deben ser datos *hashables*.

Por tanto, no se pueden usar como clave una lista, un diccionario, un conjunto `set` o cualquier otro tipo mutable.

Los tipos numéricos que se usen como claves obedecen las reglas normales de comparación numérica.

- Por tanto, si dos números son considerados iguales (como `1` y `1.0`) entonces se consideran la misma clave en el diccionario.

Para **acceder a un elemento** del diccionario se usa una sintaxis idéntica a la de la **indexación**, salvo que, en este caso, en lugar de usar el índice o posición del elemento, se usa la clave:

```
>>> a = {'perro': 'dog', 'gato': 'cat'}
>>> a['perro']
'dog'
```

Si se intenta acceder a un elemento usando una clave que no existe, se lanza una excepción de tipo `KeyError`:

```
>>> a['caballo']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'caballo'
```

3.2.1. Operaciones

d y *o* son diccionarios, *c* es una clave válida y *v* es un valor cualquiera:

Operación	Resultado
`d[c] = v`	Asigna a `d[c]` el valor *v*
`del d[c]`	Borra `d[c]` de *d* (lanza `KeyError` si *c* no está en *d*)
`c in d`	`True` si *d* contiene una clave *c*
`c not in d`	`True` si *d* no contiene una clave *c*
`d.clear()`	Elimina todos los elementos de *d*
`d.copy()`	Devuelve una copia superficial de *d*
`d.get(c[, def])`	Devuelve el valor de *c* si *c* está en *d*; en caso contrario, devuelve *def* (que por defecto es `None`)
`d.pop(c[, def])`	Elimina y devuelve el valor de *c* si *c* está en *d*; en caso contrario, devuelve *def* (si no se pasa *def* y *c* no está en *d*, produce un `KeyError`)

Operación	Resultado
d.popitem()	Elimina y devuelve una pareja (*clave*, *valor*) del diccionario siguiendo un orden LIFO (produce un KeyError si *d* está vacío)
d.setdefault(*c*[, *def*])	Si *c* está en *d*, devuelve su valor; si no, inserta *c* en *d* con el valor *def* y devuelve *def* (por defecto, *def* vale None)
d.update(*o*)	Actualiza *d* con las parejas (*clave*, *valor*) de *o*, sobreescribiendo las claves ya existentes, y devuelve None

3.2.2. Recorrido de diccionarios

Como cualquier otro dato iterable, los diccionarios se pueden recorrer usando iteradores.

Los iteradores creados sobre un diccionario, en realidad, recorren sus **claves**:

```
>>> d
{'a': 1, 'b': 2}
>>> it = iter(d)
>>> next(it)
'a'
>>> next(it)
'b'
```

Lo mismo ocurre con un bucle **for**:

```
>>> for k in d:
...     print(k)
...
a
b
```

Si necesitamos acceder también a los valores de un diccionario mientras lo recorremos con un bucle **for**, tenemos dos opciones:

- Acceder al valor a partir de la clave usando **indexación**:

  ```
  >>> for k in d:
  ...     print(k, d[k])
  ...
  a 1
  b 2
  ```

- Usar el método items sobre el diccionario (el cual devuelve un objeto que, al iterar sobre él, genera una secuencia de tuplas **(**⟨*clave*⟩**,** ⟨*valor*⟩**)**), combinándolo con el **desempaquetado de tuplas**:

  ```
  >>> d.items()
  dict_items([('a', 1), ('b', 2)])
  >>> for k, v in d.items():
  ```

```
...     print(k, v)
...
a 1
b 2
```

Otros métodos útiles para recorrer un diccionario son `keys` y `values`.

`keys` devuelve un objeto que, al iterar sobre él, va generando las **claves** del diccionario sobre el que se invoca:

```
>>> d.keys()
dict_keys(['a', 'b'])
>>> for k in d.keys():
...     print(k)
...
a
b
```

`values` devuelve un objeto que, al iterar sobre él, va generando los **valores** del diccionario sobre el que se invoca:

```
>>> d.values()
dict_values([1, 2])
>>> for v in d.values():
...     print(v)
...
1
2
```

En la práctica, no resulta muy útil usar `keys`, ya que se puede hacer lo mismo recorriendo directamente el propio diccionario, como ya sabemos:

```
>>> for k in d:
...     print(k)
...
a
b
```

¿En qué orden se almacenan los elementos de un diccionario? Y, lo más importante: ¿en qué orden se van visitando los elementos de un diccionario cuando se recorre con un iterador?

- Antes de la versión 3.7 de Python, no estaba determinado en qué orden se almacenaban los elementos en un diccionario. Por tanto, al recorrer un diccionario con un iterador, no se sabía de antemano en qué orden se iban a visitar sus elementos.
- A partir de la versión 3.7, se decidió que los elementos se almacenarían en el orden en el que se van insertando dentro del diccionario.

Por tanto:

> A partir de Python 3.7, al recorrer un diccionario, un iterador devolverá siempre sus elementos **en el orden en el que se han ido insertando** dentro del diccionario.

3.3. Documentos XML

Los documentos XML se pueden considerar **datos estructurados en forma de *árbol*** (es decir, con una estructura **jerárquica** y, por tanto, **no secuencial**).

Por ejemplo, supongamos el siguiente documento XML:

```xml
<?xml version="1.0"?>
<raiz>
    <alumno numero="111">
        <dni>12312312A</dni>
        <nombre>
            <propio>Juan</propio>
            <apellidos>García González</apellidos>
        </nombre>
        <telefono>666555444</telefono>
        <nota>7</nota>
    </alumno>
    <alumno numero="222">
        <dni>44455566B</dni>
        <nombre>
            <propio>María</propio>
            <apellidos>Pérez Rodríguez</apellidos>
        </nombre>
        <telefono>696969696</telefono>
        <nota>9</nota>
    </alumno>
    <madre>
        <dni>22222222C</dni>
    </madre>
</raiz>
```

Podemos encontrarnos lo siguiente:

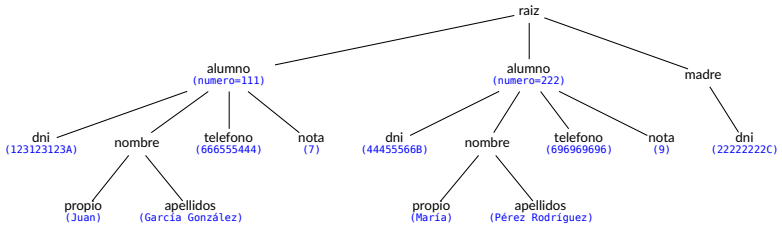
- Lo que hay entre ángulos (como `<raiz>`) es una *etiqueta*.
- Lo que hay entre dos etiquetas (como el `7` en `<nota>7</nota>`) es el *texto* de la etiqueta.
- Lo que hay dentro de la etiqueta (como `numero="111"`) es un *atributo* de la etiqueta.

  `numero="111"` es un atributo de la primera etiqueta `<alumno>`.

  `numero` es el *nombre* del atributo.

  `111` es el *valor* del atributo `numero`.
- Puede haber etiquetas con uno o varios atributos y etiquetas sin atributos.

Ese documento representaría la siguiente estructura jerárquica:



Se observa que:

- **Cada nodo representa una etiqueta** del documento XML.
- Si una etiqueta contiene a otra, su correspondiente nodo en el árbol tendrá un hijo que representa a la etiqueta que contiene.
- Los hijos de un nodo están ordenados por la posición que ocupan dentro de su etiqueta padre.

3.3.1. Acceso

El módulo `xml.etree.ElementTree` (documentado en https://docs.python.org/3/library/xml.etree.elementtree.html) implementa una interfaz sencilla y eficiente para interpretar y crear datos XML.

Para importar los datos de un archivo XML, podemos hacer:

```
import xml.etree.ElementTree as ET
arbol = ET.parse('archivo.xml')
raiz = arbol.getroot()
```

Si los datos XML se encuentran ya en una cadena, se puede hacer directamente:

```
raiz = ET.fromstring(datos_en_una_cadena)
```

Los nodos del árbol se representan internamente mediante objetos de tipo `Element`, los cuales disponen de ciertos atributos y responden a ciertos métodos.

¡Cuidado! Aquí, cuando hablamos de *atributos*, nos referimos a información que contiene el objeto (una cualidad del objeto según el *paradigma orientado a objetos*) y a la cual se puede acceder usando el operador punto (`.`), no a los atributos que pueda tener una etiqueta según consten en el documento XML.

Los objetos de tipo `Element` disponen de los siguientes atributos:

- `tag`: una cadena que representa a la etiqueta del nodo (por ejemplo: si la etiqueta es `<alumno>`, entonces `tag` contendrá `'alumno'`).
- `attrib`: un diccionario que representa a los atributos de esa etiqueta en el documento XML.
- `text`: una cadena que representa el contenido del nodo, es decir, el texto que hay entre ***<etiqueta>*** y ***</etiqueta>***.

Por ejemplo, si tenemos la siguiente etiqueta en el documento XML:

```
<telefono tipo="movil">666555444</telefono>
```

y la variable `nodo` contiene el nodo (es decir, el objeto `Element`) que representa a dicha etiqueta en el árbol, entonces:

- `nodo.tag` valdrá `'telefono'`.
- `nodo.attrib` valdrá `{'tipo': 'movil'}`.
- `nodo.text` valdrá `'666555444'`.

En nuestro caso, `raiz` es un objeto de tipo `Element` que, además, representa al nodo raíz del árbol XML.

Por tanto, tendríamos lo siguiente:

```
>>> raiz.tag
'raiz'
>>> raiz.attrib
{}
>>> raiz.text
'\n    '              # ¿Por qué?
```

Los objetos `Element` son **iterables**. Por ejemplo, el nodo raíz tiene **nodos hijos** (nodos que «cuelgan» directamente del nodo raíz) sobre los cuales se puede iterar desde el objeto `raiz`:

```
>>> for hijo in raiz:
...     print(hijo.tag, hijo.attrib)
...
alumno {'numero': '111'}
alumno {'numero': '222'}
madre {}
```

Los hijos están anidados, y podemos acceder a nodos concretos a través de su índice (es decir, que los objetos `Element` son **indexables**):

```
>>> raiz[0]       # el primer hijo directo de raiz
<Element 'alumno' at 0x7f929c29cf90>
>>> raiz[0][2]    # el tercer hijo directo del primer hijo directo de raiz
<Element 'telefono' at 0x7f929c29d180>
>>> raiz[0][2].text
'666555444'
```

Los objetos de tipo `Element` disponen de métodos útiles para iterar recursivamente sobre todos los subárboles situados debajo de él (sus hijos, los hijos de sus hijos, y así sucesivamente).

Por ejemplo, el método `iter` devuelve un **iterador** que recorre todos los nodos del árbol desde el nodo actual (el nodo sobre el que se invoca al método) en un orden *primero en profundidad*.

Eso quiere decir que va visitando los nodos en el mismo orden en el que se encuentran escritos en el documento XML, incluyendo el propio nodo sobre el que se invoca.

Por ejemplo:

```
>>> for nodo in raiz.iter():
...     print(nodo.tag)
...
raiz
alumno
dni
nombre
propio
apellidos
telefono
nota
```

```
alumno
dni
nombre
propio
apellidos
telefono
nota
madre
dni
```

Si se le pasa una etiqueta como argumento, devolverá únicamente los nodos que tengan esa etiqueta:

```
>>> for dni in raiz.iter('dni'):
...     print(dni.text)
...
12312312A
44455566B
22222222C
```

El método `findall` devuelve una lista con los nodos que tengan una cierta etiqueta y que sean hijos directos del nodo sobre el que se invoca.

Puede devolver una lista vacía si no hay nodos que cumplan la condición.

El método `find` devuelve el primer hijo directo del nodo sobre el que se invoca, siempre que tenga una cierta etiqueta indicada como argumento.

Puede devolver `None` si el nodo no tiene ningún hijo con esa etiqueta.

El método `get` devuelve el valor de algún atributo de la etiqueta asociada a ese nodo:

```
>>> for alumno in raiz.findall('alumno'):
...     numero = alumno.get('numero')
...     dni = alumno.find('dni').text
...     print(numero, dni)
...
111 12312312A
222 44455566B
```

Si la etiqueta no tiene el atributo indicado en el argumento de `get`, éste devuelve `None` o el valor que se haya indicado en el segundo argumento:

```
>>> alumno = raiz[0]
>>> alumno.get('numero')
111
>>> alumno.get('altura')
None
>>> alumno.get('altura', 0)
0
>>> alumno.get('numero', 0)
111
```

También disponemos de la función `dump` que devuelve la cadena correspondiente al nodo que se le pase como argumento:

```
>>> ET.dump(raiz[0][2])
<telefono>666555444</telefono>
```

Para una especificación más sofisticada de los elementos a encontrar, se pueden usar las expresiones `XPath` con los métodos `find` y `findall`:

Sintaxis	Significado
etiqueta	Selecciona todos los nodos hijo con la etiqueta ***etiqueta***. Por ejemplo, `pepe` selecciona todos los nodos hijo llamados `pepe`, y `pepe/juan` selecciona todos los nietos llamados `juan` en todos los hijos llamados `pepe`.
`*`	Selecciona todos los nodos hijo inmediatos. Por ejemplo, `*/pepe` selecciona todos los nietos llamados `pepe`.
`.`	Selecciona el nodo actual. Se usa, sobre todo, al principio de la ruta para indicar que es una ruta relativa.
`//`	Selecciona todos los subnodos en cualquier nivel por debajo de un nodo. Por ejemplo, `.//pepe` selecciona todos los nodos `pepe` que haya en todo el árbol.
`..`	Selecciona el nodo padre. Devuelve `None` si se intenta acceder a un ancestro del nodo de inicio (aquel sobre el que se ha llamado al método `find`).

Continuación de las expresiones `XPath`:

Sintaxis	Significado
`[@`***atrib***`]`	Selecciona todos los nodos que tienen el atributo ***atrib***.
`[@`***atrib***`='`***valor***`']`	Selecciona todos los nodos que tienen el valor ***valor*** en el atributo ***atrib***. El valor no puede contener apóstrofes.
`[@`***atrib***`!='`***valor***`']`	Selecciona todos los nodos que tienen el valor ***valor*** en el atributo ***atrib***. El valor no puede contener apóstrofes.
`[`***etiqueta***`]`	Selecciona todos los nodos que tienen un hijo inmediato llamado ***etiqueta***.
`[`***posición***`]`	Selecciona todos los nodos situados en cierta ***posición***. Ésta puede ser un entero (`1` es la primera posición), la expresión `last()` (la última posición) o una posición relativa a la última posición (por ejemplo, `last() - 1`).

Ejemplos

```
# Los nodos de nivel más alto:
>>> raiz.findall(".")
[<Element 'raiz' at 0x7f929c29cf40>]

# Los nietos 'dni' de los hijos 'alumno' de los nodos de nivel más alto:
>>> raiz.findall("./alumno/dni")
[<Element 'dni' at 0x7f929c29d040>,
 <Element 'dni' at 0x7f929c29d270>]

# Lo de antes equivale a hacer (porque el nodo actual es el raíz):
>>> raiz.findall("alumno/dni")
[<Element 'dni' at 0x7f929c29d040>,
 <Element 'dni' at 0x7f929c29d270>]

# Los nodos con numero='111' que tienen un hijo 'dni':
>>> raiz.findall(".//dni/..[@numero='111']")
[<Element 'alumno' at 0x7f929c29cf90>]

# Antes de // hay que poner algo que indique el nodo debajo del
# cual se va a buscar:
>>> raiz.findall("//dni/..[@numero='111']")
SyntaxError: cannot use absolute path on element
```

Ejemplos

```
# Todos los nodos 'dni' del árbol completo:
>>> raiz.findall('.//dni')
[<Element 'dni' at 0x7f547a0e9950>,
 <Element 'dni' at 0x7f547a0e9b80>,
 <Element 'dni' at 0x7f547a0e9e00>]

# Sólo los DNIs que estén por debajo de un nodo 'madre'
# en cualquier nivel:
>>> raiz.findall('madre//dni')
[<Element 'dni' at 0x7f547a0e9e00>]

# Los nodos 'dni' que son hijos de los nodos con numero='111':
>>> raiz.findall(".//*[@numero='111']/dni")
[<Element 'dni' at 0x7f929c29d040>]

# Los nodos 'alumno' que son hijos segundos de sus padres:
>>> raiz.findall(".//alumno[2]")
[<Element 'alumno' at 0x7f929c29d220>]

# Los nodos 'alumno' hijos directos del actual que tienen un hijo 'nota':
>>> raiz.findall("./alumno[nota]")
[<Element 'alumno' at 0x7f929c29cf90>,
 <Element 'alumno' at 0x7f929c29d220>]

# Lo de antes equivale a hacer (porque el nodo actual es el raíz):
>>> raiz.findall("alumno[nota]")
[<Element 'alumno' at 0x7f929c29cf90>,
 <Element 'alumno' at 0x7f929c29d220>]
```

3.3.2. Modificación

`ElementTree` proporciona una forma sencilla de crear documentos XML y escribirlos en archivos.

Para ello, usamos el método `write`.

Una vez creado, un objeto `Element` puede manipularse directamente:

- Cambiando los atributos del objeto, como `text` o `attrib`.
- Cambiando los atributos de la etiqueta a la que representa el objeto, con el método `set`.
- Añadiendo nuevos hijos con los métodos `append` o `insert`.
- Eliminando hijos con el método `remove`.

Por ejemplo, supongamos que queremos sumarle 1 a la `nota` de cada alumno y añadir un atributo `modificado` a la etiqueta `nota`:

```
>>> for nota in raiz.iter('nota'):
...     nueva_nota = int(nota.text) + 1
...     nota.text = str(nueva_nota)
...     nota.set('modificado', 'si')
...
>>> arbol.write('salida.xml')
```

Nuestro XML tendría ahora el siguiente aspecto:

```
<?xml version="1.0"?>
<raiz>
    <alumno numero="111">
        <dni>12312312A</dni>
        <nombre>
            <propio>Juan</propio>
            <apellidos>García González</apellidos>
        </nombre>
        <telefono>666555444</telefono>
        <nota modificado="si">8</nota>
    </alumno>
    <alumno numero="222">
        <dni>44455566B</dni>
        <nombre>
            <propio>María</propio>
            <apellidos>Pérez Rodríguez</apellidos>
        </nombre>
        <telefono>696969696</telefono>
        <nota modificado="si">10</nota>
    </alumno>
    <madre>
        <dni>22222222C</dni>
    </madre>
</raiz>
```

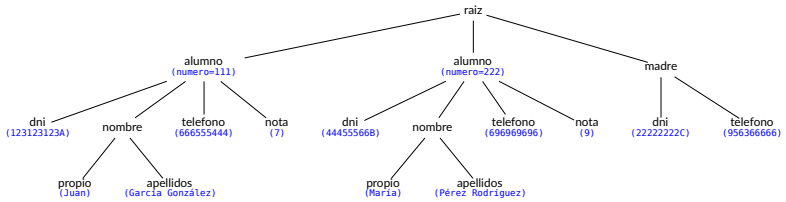
Podemos insertar nuevos nodos hijos de un determinado nodo con los métodos `append` o `insert`, como si el nodo fuese una lista.

Para ello, primero normalmente crearemos el nodo que vamos a insertar usando **`Element(`*etiqueta*`)`**.

Por ejemplo, añadimos el teléfono a la única madre que tenemos en el documento XML:

```
telefono = ET.Element('telefono')
telefono.text = '956366666'
madre = raiz.find('madre')
madre.append(telefono)
```

Tras estas operaciones, ahora tendríamos:

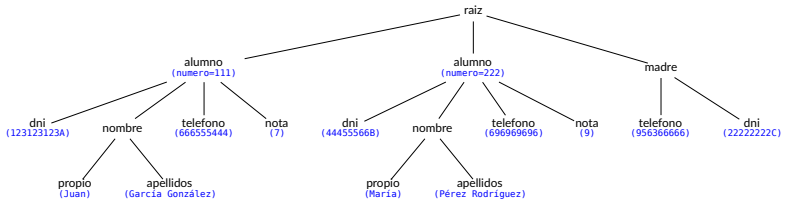


Podríamos haber usado `insert` en lugar de `append` para cambiar la posición donde situar el nodo hijo.

Por ejemplo, si queremos situar el teléfono antes que el DNI, podríamos hacer:

```
telefono = ET.Element('telefono')
telefono.text = '956366666'
madre = raiz.find('madre')
madre.insert(0, telefono)
```

Tras estas operaciones, ahora tendríamos:



Podemos eliminar elementos con el método `remove`. Por ejemplo, supongamos que queremos eliminar todos los alumnos con una `nota` inferior a 9:

```
>>> for alumno in raiz.findall('alumno'):
...     # se usa findall para que no afecte el borrado durante el recorrido
...     nota = int(alumno.find('nota').text)
...     if nota < 9:
...         raiz.remove(alumno)
...
>>> arbol.write('salida.xml')
```

Tener en cuenta que la modificación concurrente mientras se hace una iteración puede dar problemas, lo mismo que ocurre cuando se modifican listas o diccionarios mientras se itera sobre ellos.

Por ello, el ejemplo primero recoge todos los elementos con `findall` y sólo entonces itera sobre la lista que devuelve.

Si usáramos `iter` en lugar de `findall` se podrían dar problemas debido a que `iter` va devolviendo perezosamente los nodos (es un iterador) y el conjunto de nodos que devuelve podría verse afectado

por los borrados.

Nuestro XML tendría ahora el siguiente aspecto:

```
<?xml version="1.0"?>
<raiz>
    <alumno numero="222">
        <dni>44455566B</dni>
        <nombre>
            <propio>María</propio>
            <apellidos>Pérez Rodríguez</apellidos>
        </nombre>
        <telefono>696969696</telefono>
        <nota modificado="si">10</nota>
    </alumno>
    <madre>
        <dni>22222222C</dni>
    </madre>
</raiz>
```

Si lo que queremos es **mover** un nodo (es decir, cambiar un nodo de sitio), podemos combinar los efectos de `append` e `insert` con `remove`.

Por ejemplo, si queremos mover la etiqueta `<madre>` dentro de la etiqueta `<alumno>`, podríamos hacer:

```
madre = raiz.find('madre')
raiz.remove(madre)
alumno = raiz.find('alumno')
alumno.append(madre)
```

Nuestro XML tendría ahora el siguiente aspecto:

```
<?xml version="1.0"?>
<raiz>
    <alumno numero="222">
        <dni>44455566B</dni>
        <nombre>
            <propio>María</propio>
            <apellidos>Pérez Rodríguez</apellidos>
        </nombre>
        <telefono>696969696</telefono>
        <nota modificado="si">10</nota>
        <madre>
            <dni>22222222C</dni>
        </madre>
    </alumno>
</raiz>
```

La función `SubElement` también proporciona una forma muy conveniente de crear sub-elementos de un elemento dado:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
```

```
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

Bibliografía

Python Software Foundation. n.d. “Sitio Web de Documentación de Python.” https://docs.python.org/3.