Procesamiento de datos de alto nivel

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/12/12 a las 15:59:00

Índice

1. Colecciones

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

- *Componer* consiste en combinar elementos entre sí para formar otros más complejos.
- *Abstraer* consiste en coger un elemento (normalmente complejo), darle un nombre y ocultar sus detalles internos (es decir, los elementos que lo componen) dentro de una caja negra.

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

- Podemos aplicar la composición para crear nuevos elementos complejos a partir de otros elementos complejos.
- Podemos aplicar la abstracción para crear nuevas 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 la composición de datos y dejaremos la abstracción de datos 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**, los cuales representan su **contenido**.

Por contra, los datos no estructurados se denominan **datos elementales**, **escalares** o **atómicos**.

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** esos elementos 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.

En Python, el término más utilizado es el de ***colección***.

Una **colección** en Python es un objeto que **almacena o genera múltiples elementos en memoria** y provee una **interfaz estandarizada** para acceder, recorrer y consultar sus elementos.

Entre otras operaciones, una colección admite la función `len` para consultar el número de elementos que contiene.

Además, dispone de otras operaciones dependiendo del tipo concreto de colección que se trate.

1.3. Clasificación

Los **datos estructurados** se pueden clasificar atendiendo a su *secuencialidad* y a su *mutabilidad*.

Según su **secuencialidad**:

- **Secuencias:** Son aquellos en los que se puede acceder directamente y de forma eficiente a cada uno de sus elementos indicando la posición que ocupan dentro de la secuencia.

  Por tanto, son colecciones *ordenadas* por posición, ya que sus elementos están ordenados dentro de la secuencia según la posición en la que se encuentran situados dentro de la misma.
- **No secuenciales:** Son aquellos en los que **NO** se puede acceder directamente y de forma eficiente a cada uno de sus elementos indicando la posición que ocupan dentro de la colección.

  En general, las estructuras no secuenciales son colecciones *desordenadas*, en las que no se puede afirmar que sus elementos se encuentran en una posición determinada dentro de la colección.

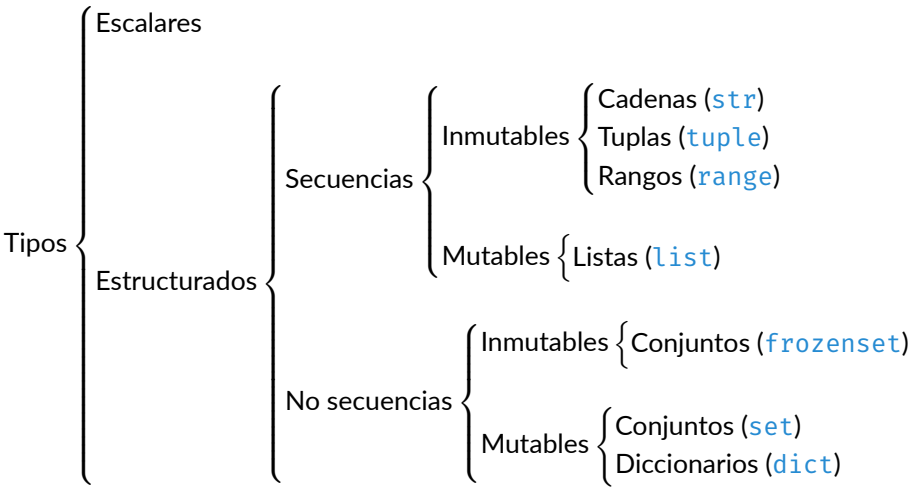
Según su **mutabilidad**:

- **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.

El **contenido** de un dato estructurado forma parte del **estado interno** de ese dato, por lo que cambiar el contenido de un dato estructurado supone cambiar también su estado interno.

Por ejemplo, si en la lista `[7, 8, 9]` sustituimos su segundo elemento (el `8`) por un `5` para obtener la lista `[7, 5, 9]`, estamos cambiando el contenido de la lista y, por consiguiente, su estado interno.

Su identidad no ha cambiado, pero su estado interno sí.



1.4. *Hashables*

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

1. Puede compararse con otros datos usando el operador `==`.
2. Tiene asociado un número entero llamado ***hash*** que nunca cambia durante toda la vida del dato.

   Para obtener el *hash* de un dato, se usa la función `hash`:

   - Si un dato *d* es *hashable*, `hash(d)` devolverá el *hash* de *d*.
   - En caso contrario, lanzará una excepción de tipo `TypeError`.

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

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

En cambio, si dos datos son distintos, sus *hash* no tienen por qué serlo.

Ejemplos:

```
>>> hash('hola')
6290906884732116299
>>> hash('hola')
6290906884732116299
>>> 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'
```

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

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

La mayoría de los datos inmutables predefinidos en Python son *hashables*, incluyendo los valores de tipo `int`, `float`, `str`, `range` y `bool`, así como las funciones y el valor `None`.

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

Las **colecciones mutables** (como las listas o los diccionarios) **NO** son *hashables*.

El *hash* de un dato depende del estado interno del dato, ya que se calcula a partir de dicho estado interno usando un algoritmo que no nos debe preocupar por ahora.

Como el estado interno de una colección viene determinado principalmente por los elementos que contiene, el *hash* de una colección dependerá también del contenido de la colección.

Y por esta razón, las colecciones mutables no son *hashables*: si una colección es mutable, su contenido puede cambiar y, por tanto, su *hash* también cambiaría, pero esto está prohibido.

El *hash* de un dato **se calcula** en función del **estado interno** del dato y, en caso de ser una colección, también en función de su **contenido**.

El *hash* de un dato es un número que **representa al dato y a todo su contenido**.

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

El *hash* de un dato se utiliza internamente para acceder a ese dato dentro de una colección de forma directa y eficiente.

Para ello, el intérprete utiliza ciertas técnicas que permiten localizar directamente a un dato dentro de una colección, de forma casi inmediata y sin importar el tamaño de la colección (pero recordemos que para ello es necesario que el *hash* del dato nunca cambie).

De no usar estas técnicas, el intérprete tendría que buscar el dato secuencialmente dentro de la colección, recorriéndola desde el principio hasta el final, lo que sería mucho más lento y consumiría un tiempo que sería mayor cuanto más grande fuese la colección.

Los *hash* **permiten el acceso *directo* a un dato** dentro de una colección.

Muy en resumen, esas técnicas se basan en dividir el espacio de memoria que ocupa la colección en una serie de *almacenes* 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 inmediata, con coste $O(1)$).
3. Localizar dentro del *bucket* el elemento que se está buscando usando el `==`, lo cual consumirá un tiempo que, en general, no será mucho, ya que los elementos están repartidos entre todos los *buckets* y, por tanto, normalmente no habrá muchos elementos en cada *bucket*.

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

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

Función `id`	Función `hash`
Devuelve la identidad de un dato.	Devuelve el *hash* de un dato, si es *hashable*.
Todos los datos tienen identidad.	No todos los datos son *hashables*.
Puede haber datos iguales pero no idénticos.	Si dos datos son iguales, sus *hash* también deben serlo.
Su valor no depende del estado interno del dato y, por tanto, tampoco de su contenido.	Su valor se obtiene a partir del estado interno del dato (y, por tanto, de su contenido), usando una fórmula matemática.
Por tanto, no cambia si se modifica el dato.	Por tanto, un dato mutable no puede ser *hashable*, ya que su *hash* cambiaría al cambiar su contenido o estado interno.

2. Flujos perezosos

2.1. Iterables e iteradores

Se dice que un dato compuesto es **iterable** cuando se puede acceder a todos sus elementos de uno en uno, operación que se denomina **recorrer** el iterable.

Gracias a esto, se dice que un iterable nos permite *visitar* sus elementos o, también, *iterar* sobre sus elementos.

Como iterables tenemos:

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

Los iterables no representan un tipo concreto, sino más bien una *familia* de tipos que comparten la misma propiedad.

Muchas funciones actúan sobre iterables en general, en lugar de hacerlo sobre un tipo estructurado concreto (lista, tupla, ...).

Por ejemplo, las listas nos permiten acceder a todos sus elementos de uno en uno y, por tanto, podemos recorrerla.

Para *visitar* sus elementos podemos usar la *indexación*, y para *recorrer* toda la lista podemos usar un *bucle*:

```
def recorrer_lista(l):
    i = 0
    while i < len(l):
        print(l[i])
        i += 1
```

Pero hay que hacerlo bien para asegurarse de que no se visita el mismo elemento dos veces, o se quedan elementos sin visitar, o se intenta acceder a un elemento inexistente a través de un índice que está fuera del rango permitido por la lista.

La manera general de recorrer un dato iterable es usando un **iterador**.

De hecho, técnicamente, **un *iterable* se define** como aquel dato al que le podemos asociar, al menos, un ***iterador***.

Un **iterador** es un objeto que sabe cómo **recorrer** un iterable concreto.

Para ello, el iterador crea un **flujo de datos *perezoso*** que va entregando los elementos de ese iterable de uno en uno a medida que se los vamos solicitando.

Los sucesivos elementos del flujo de datos se van obteniendo al llamar repetidamente a la función `next` aplicada al iterador.

Cuando ya no hay más elementos disponibles, la función `next` lanza una excepción de tipo `StopIteration`, lo que indica que el iterador **se ha agotado** (se han consumido todos sus elementos), por lo que si se sigue llamando a la función `next` se seguirá lanzando esa excepción.

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

(Recordemos que todo iterable puede tener asociado un iterador.)

Ejemplo de uso de `iter` y `next`:

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

Si se le pasa un dato no iterable, `iter` lanza una excepción `TypeError`:

```
>>> it = iter(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Los iteradores son **iterables perezosos de un solo uso**:

- Son **perezosos** porque van generando sus elementos a medida que los va entregando, en lugar de generarlos todos a la vez primero.
- Son **de un solo uso** porque cada elemento sólo se entrega una vez.

Además, **los iteradores 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>
>>> it is it2
True
```

- En consecuencia, podemos usar un iterador en cualquier sitio donde se espere un iterable.

Muchas funciones devuelven iteradores en lugar de colecciones porque, al ser perezosos, los iteradores son más eficientes en memoria que si se devolviera toda una lista o tupla.

El flujo de datos que entrega un iterador se puede convertir en una lista o tupla usando las funciones `list` y `tuple`:

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

2.2. El bucle `for`

Probablemente, la mejor forma de recorrer los elementos que entrega 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⟩)
while True:
    try:
        ⟨variable⟩(, ⟨variable⟩)* = next(iterador)
    except StopIteration:
        break
    else:
        ⟨sentencia⟩
```

Ejemplo:

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

devuelve:

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

2.3. range

Los **rangos** (valores de tipo `range`) representan **secuencias perezosas, inmutables y *hashables* de números enteros** situados dentro de un intervalo.

Los rangos se crean con la función `range`, cuya signatura es:

```
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.

Por ejemplo:

- `range(2, 7)` representa la secuencia de números enteros comprendidos entre 2 y 6, es decir, la secuencia 2, 3, 4, 5, 6.
- `range(2, 7, 2)` representa la misma secuencia pero saltándose uno cada vez, es decir, la secuencia 2, 4, 6.
- `range(3)` representa la secuencia 0, 1, 2 (se empieza por 0 si no se indica otra cosa).
- `range(4, 0, -1)` representa la secuencia 4, 3, 2, 1.

Como ocurre en toda secuencia, podemos usar ***indexación*** para acceder al contenido de un rango, es decir, a cada uno de sus elementos.

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 son perezosos ya que no hay que reservar espacio en memoria para almacenar todos sus elementos simultáneamente, sino que tan sólo necesita recordar los valores de *start*, *stop* y *step*.

Esto hace que ocupen mucho menos espacio en memoria que las listas o las tuplas, por ejemplo.

Los rangos, además, son iterables, por lo que se pueden usar iteradores para recorrer sus elementos.

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))
[]
```

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)
```

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
```

Los rangos se usan frecuentemente para hacer bucles que se repitan un determinado número de veces.

Por ejemplo, el cuerpo del siguiente bucle se ejecuta 5 veces:

```
for i in range(5):
    print(i)
```

y produce la siguiente salida:

```
0
1
2
3
4
```

2.4. Funciones que devuelven iteradores

2.4.1. El módulo `itertools`

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

Algunos de esos iteradores son muy especiales porque pueden entregar flujos infinitos o valores que se repiten continuamente, lo cual contradice en cierta manera lo que dijimos cuando definimos los iteradores como «*iterables de un solo uso*».

`itertools.count(`[⟨*inicio*⟩[`,` ⟨*paso*⟩]]`)` entrega un flujo infinito de valores separados uniformemente. Se puede indicar opcionalmente un valor de comienzo (⟨*inicio*⟩, que por defecto es `0`) y el intervalo entre números (⟨*paso*⟩, 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(`⟨*iterable*⟩`)` devuelve un iterador que va entregando los elementos del iterable del primero al último, repitiéndolos indefinidamente:

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

`itertools.repeat(`⟨*elem*⟩[`,` ⟨*n*⟩]`)` entrega ⟨*n*⟩ veces el elemento ⟨*elem*⟩, o lo entrega 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

2.4.2. `enumerate`

Si estamos recorriendo una secuencia y necesitamos recuperar tanto el valor como el **índice** de cada elemento, podemos usar la función `enumerate`.

Esta función devuelve un iterador que va entregando tuplas que contienen, además del elemento, el valor correspondiente de un contador numérico.

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

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

```
>>> 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
```

2.4.3. `zip`

La función `zip` en Python devuelve un iterador de tuplas, donde cada tupla contiene un elemento de cada uno de los iterables que se le pasen como argumentos, agrupados por su posición.

En otras palabras:

- Toma los elementos en posición 0 de cada iterable y forma la primera tupla.
- Luego toma los elementos en posición 1 de cada iterable y forma la segunda tupla.
- Y así sucesivamente, hasta que el iterable más corto se quede sin elementos.

Su sintaxis es:

```
zip(⟨iterable_1⟩ [, ... ⟨iterable_n⟩])
```

Ejemplos

El siguiente código:

```
a = [1, 2, 3]
b = ['x', 'y', 'z']
c = zip(a, b)

print(list(c))
```

produce la siguiente salida:

```
[(1, 'x'), (2, 'y'), (3, 'z')]
```

Si los iterables tienen distinta longitud, la función `zip` se detiene en el más corto. Por ejemplo, el siguiente código:

```
a = [1, 2]
b = ['x', 'y', 'z']
print(list(zip(a, b)))
```

produce la siguiente salida:

```
[(1, 'x'), (2, 'y')]
```

2.4.4. `reversed`

La función `reversed` en Python devuelve un iterador que entrega los elementos de un iterable en orden inverso a como se entregarían habitualmente.

Su sintaxis es:

```
reversed(⟨iterable⟩)
```

Por ejemplo, el siguiente código:

```
l = [1, 2, 3]
it = reversed(l)

print(list(it))
```

produce la siguiente salida:

```
[3, 2, 1]
```

3. Funciones de orden superior

3.1. Concepto

Sabemos que, en programación funcional, **las *funciones* también son *valores***.

Por tanto, como pasa con cualquier otro valor, las funciones también tienen un tipo, se pueden ligar a identificadores, etcétera.

Pero si las funciones son valores, eso significa que también se pueden pasar como argumentos a otras funciones o se pueden devolver como resultado de otras funciones.

Una **función de orden superior** es una función que recibe funciones como argumentos o devuelve funciones como resultado.

Por ejemplo, la siguiente función **recibe otra función como argumento** y devuelve el resultado de aplicar dicha función al número 5:

```
>>> aplica5 = lambda f: f(5)
>>> cuadrado = lambda x: x ** 2
>>> cubo = lambda x: x ** 3
>>> aplica5(cuadrado)
25
>>> aplica5(cubo)
125
```

No hace falta crear las funciones `cuadrado` y `cubo` para pasárselas a la función `aplica5` como argumento. Se pueden pasar directamente las expresiones lambda, que también son funciones:

```
>>> aplica5(lambda x: x ** 2)
25
>>> aplica5(lambda x: x ** 3)
125
```

Naturalmente, la función que se pasa a `aplica5` debe recibir un único argumento de tipo numérico.

Lo mismo se puede hacer con funciones imperativas, o combinando funciones imperativas con expresiones lambda:

```
>>> def aplica5(f):
...     return f(5)
>>> def cuadrado(x):
...     return x ** 2
>>> cubo = lambda x: x ** 3
>>> aplica5(cuadrado)
25
>>> aplica5(cubo)
125
```

También se puede **devolver una función como resultado**.

Por ejemplo, la siguiente función `suma_o_resta` recibe una cadena y devuelve una función que suma si la cadena es `'suma'`; en caso contrario, devuelve una función que resta:

```
>>> suma_o_resta = lambda s: (lambda x, y: x + y) if s == 'suma' else \
                             (lambda x, y: x - y)
>>> suma_o_resta('suma')
<function <lambda>.<locals>.<lambda> at 0x7f526ab4a790>
>>> suma = suma_o_resta('suma')
>>> suma(2, 3)
5
>>> resta = suma_o_resta('resta')
>>> resta(4, 3)
1
>>> suma_o_resta('suma')(6, 4)
10
```

Tanto `aplica5` como `suma_o_resta` son **funciones de orden superior**.

Y con funciones imperativas:

```
>>> def suma_o_resta(s):
...     def sumar(x, y):
...         return x + y
...     def restar(x, y):
...         return x - y
...     if s == 'suma':
...         return sumar
...     else:
...         return restar
>>> suma_o_resta('suma')
<function <lambda>.<locals>.sumar at 0x7f3ab30de200>
>>> suma = suma_o_resta('suma')
>>> suma(2, 3)
5
>>> resta = suma_o_resta('resta')
>>> resta(4, 3)
1
>>> suma_o_resta('suma')(6, 4)
10
```

Una función es una abstracción porque agrupa lo que tienen en común determinados casos particulares que siguen el mismo patrón.

El mismo concepto se puede aplicar a casos particulares de funciones, y al hacerlo damos un paso más en nuestro camino hacia un mayor grado de abstracción.

Es decir: muchas veces observamos el mismo patrón en funciones diferentes.

Para poder abstraer, de nuevo, lo que tienen en común dichas funciones, deberíamos ser capaces de manejar funciones que acepten a otras funciones como argumentos o que devuelvan otra función como resultado (es decir, funciones de orden superior).

Supongamos las dos funciones siguientes:

```
def suma_enteros(a, b):
    """Suma los enteros comprendidos entre a y b."""
    suma = 0
    for i in range(a, b + 1):
        suma += i
    return suma
```

```
def suma_enteros(a, b):
    """Suma los cubos de los enteros comprendidos entre a y b."""
    suma = 0
    for i in range(a, b + 1):
        suma += cubo(i)
    return suma
```

Estas dos funciones comparten claramente un patrón común. Se diferencian solamente en:

- El *nombre* de la función.
- La función que se aplica a `a` para calcular cada *término* de la suma.

Podríamos haber escrito las funciones anteriores rellenando los «casilleros» ⟨*nombre*⟩ y ⟨*término*⟩ del siguiente *patrón general*:

```
def ⟨nombre⟩(a, b):
    suma = 0
    for i in range(a, b + 1):
        suma += ⟨término⟩(i)
    return suma
```

La existencia de este patrón común nos demuestra que hay una abstracción esperando que la saquemos a la superficie.

De hecho, los matemáticos han identificado hace mucho tiempo esta abstracción llamándola **sumatorio de una serie**, y la expresan así:

$$\sum_{n=a}^{b} f(n)$$

La ventaja que tiene usar la notación anterior es que podemos trabajar directamente con el concepto de sumatorio en vez de trabajar con sumas concretas, y podemos sacar conclusiones generales sobre los sumatorios independientemente de la serie particular con la que estemos trabajando.

Igualmente, como programadores estamos interesados en que nuestro lenguaje tenga la suficiente potencia como para describir directamente el concepto de sumatorio, en vez de funciones particulares que calculen sumas concretas.

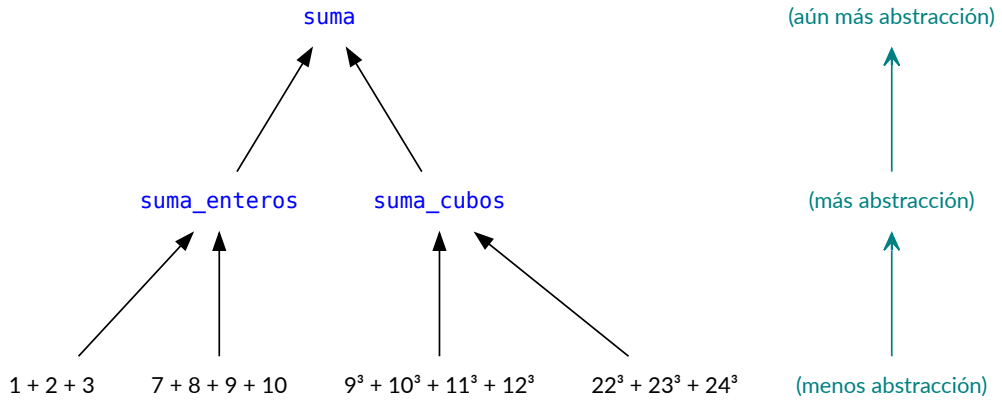
En programación lo conseguimos creando funciones que conviertan los «casilleros» en parámetros que recibirían funciones:

```
def suma(term, a, b):
    suma = 0
    for i in range(a, b + 1):
        suma += term(i)
    return suma
```

De esta forma, las dos funciones `suma_enteros` y `suma_cubos` anteriores se podrían definir en términos de esta `suma`:

```
def suma_enteros(a, b):
    return suma(lambda x: x, a, b)
def suma_cubos(a, b):
    return suma(lambda x: x * x * x, a, b)  # o suma(cubo, a, b)
```

`suma` es una abstracción que captura el patrón común que comparten `suma_enteros` y `suma_cubos`, las cuales también son abstracciones que capturan sus respectivos patrones comunes.



El camino de subida hacia una abstracción cada vez mayor

Ejercicio

1. ¿Se podría generalizar aún más la función `suma`?

3.2. map

Supongamos que queremos escribir una función que, dada una lista de números, nos devuelva otra lista con los mismos números elevados al cubo.

Ejercicio

2. Inténtalo.

Una forma de hacerlo sería:

```
def elevar_cubo(l):
    res = []
    for e in l:
        res.append(e ** 3)
    return res
```

¿Y elevar a la cuarta potencia?

```
def elevar_cuarta(l):
    res = []
    for e in l:
        res.append(e ** 4)
    return res
```

Es evidente que hay un patrón subyacente que se podría abstraer creando una función de orden superior que aplique una función `f` a los elementos de una tupla y devuelva la tupla resultante.

Esa función se llama `map`, y viene definida en Python con la siguiente signatura:

```
map(func, iterable) -> Iterator
```

donde:

- *func* debe ser una función de un solo argumento.
- *iterable* puede ser cualquier iterable.

Podemos usarla así:

```
>>> map(cubo, (1, 2, 3, 4))
<map object at 0x7f22b25e9d68>
```

Lo que devuelve es un iterador que luego podemos recorrer o, por ejemplo, convertir en una lista usando la función `list`:

```
>>> list(map(cubo, (1, 2, 3, 4)))
[1, 8, 27, 64]
```

Podemos usar cualquier iterable como argumento para `map`, como por ejemplo un rango:

```
>>> for e in map(cubo, range(1, 5)):
...     print(e, end=' ')
1 8 27 64
```

¿Cómo definirías la función `map` de forma que devolviera una lista?

Ejercicio

3. Inténtalo.

Podríamos definirla así:

```
def map(f, l):
    res = []
    for e in l:
        res.append(f(e))
    return res
```

3.3. filter

`filter` es una **función de orden superior** que devuelve aquellos elementos de un *iterable* que cumplen una determinada condición.

Su signatura es:

```
filter(function, iterable) -> Iterator
```

donde *function* debe ser una función de un solo argumento que devuelva un *booleano*.

Como `map`, también devuelve un *iterador*.

Por ejemplo:

```
>>> for e in filter(lambda x: x > 0, (-4, 3, 5, -2, 8, -3, 9)):
...     print(e, end=' ')
3 5 8 9
```

3.4. reduce

`reduce` es una **función de orden superior** que aplica, de forma *acumulativa*, una función a todos los elementos de un *iterable*.

Captura un **patrón muy frecuente** de recursión sobre secuencias.

Por ejemplo, para calcular la suma y el producto de todos los elementos de una lista, haríamos:

```
def suma(l):
    res = 0
    for e in l:
        res += e
    return res
```

```
>>> suma([1, 2, 3, 4])
10
```

```
def producto(l):
    res = 1
    for e in l:
        res *= e
    return res
```

```
>>> producto([1, 2, 3, 4])
24
```

Como podemos observar, la estrategia de cálculo es esencialmente la misma; sólo se diferencian en la *operación* a realizar (`+` o `*`) y en el *valor inicial* o *elemento neutro* (`0` o `1`).

Si abstraemos ese patrón común, podemos crear una función de orden superior que capture la idea de **reducir todos los elementos de un iterable a un único valor**.

Eso es lo que hace la función `reduce`.

Su signatura es:

```
reduce(function, sequence [, initial]) -> Any
```

donde:

- *function* debe ser una función que reciba dos argumentos.
- *sequence* debe ser cualquier objeto iterable (normalmente, una secuencia como una cadena, una tupla o un rango).
- *initial*, si se indica, se usará como primer elemento sobre el que realizar el cálculo y servirá como valor por defecto cuando la secuencia esté vacía (si no se indica y la secuencia está vacía, generará un error).

Para usarla, primero tenemos que *importarla* del *módulo* `functools`:

```
from functools import reduce
```

No es la primera vez que importamos un módulo. Ya lo hicimos con el módulo `math`.

En su momento estudiaremos con detalle qué son los módulos. Por ahora nos basta con lo que ya sabemos: que contienen definiciones que podemos incorporar a nuestros *scripts*.

Por ejemplo, para calcular la suma y el producto de `(1, 2, 3, 4)`, podemos definir las funciones `suma_de_numeros` y `producto_de_numeros` a partir de `reduce`:

```
from functools import reduce

tupla = (1, 2, 3, 4)

def suma_de_numeros(tupla):
    return reduce(lambda x, y: x + y, tupla, 0)

def producto_de_numeros(tupla):
    return reduce(lambda x, y: x * y, tupla, 1)
```

También podemos importar y usar las funciones `add` y `mul` del módulo `operator`, las cuales actúan, respectivamente, como el operador `+` y `*`:

```
from functools import reduce
from operator import add, mul

tupla = (1, 2, 3, 4)

def suma_de_numeros(tupla):
    return reduce(add, tupla, 0)

def producto_de_numeros(tupla):
    return reduce(mul, tupla, 1)
```

De esta forma, usamos `add` y `mul` en lugar de las expresiones lambda `(lambda x, y: x + y)` y

`(lambda x, y: x * y)`, respectivamente.

En general, si *iterable* representa un objeto iterable que contiene los elementos $e_1, e_2, \ldots, e_n$ (en este orden), entonces tenemos que:

$$\texttt{reduce(}f\texttt{, }\mathit{iterable}\texttt{, }\mathit{ini}\texttt{)} = f(\ldots f(f(f(\mathit{ini}, e_1), e_2), e_3), \ldots, e_n)$$

Por ejemplo, la siguiente llamada a `reduce`:

```
reduce(add, (1, 2, 3, 4), 0)
```

realiza y devuelve el resultado del siguiente cálculo:

```
add(add(add(add(0, 1), 2), 3), 4)
```

lo que, en la práctica, equivale a:

```
((((0 + 1) + 2) + 3) + 4)
```

Si *iterable* representa un **iterable vacío**, entonces:

$$\texttt{reduce(}f\texttt{, }\mathit{iterable}\texttt{, }\mathit{ini}\texttt{)} = \mathit{ini}$$

Por ejemplo:

```
reduce(add, (), 0)
```

devuelve directamente `0`.

Si **no se indica un valor inicial**, tenemos que:

$$\texttt{reduce(}f\texttt{, }(e_1, e_2, \ldots, e_n)\texttt{)} = f(\ldots f(f(e_1, e_2), e_3), \ldots, e_n)$$

Es decir: se usará el primer elemento del iterable como valor inicial.

Por ejemplo, la siguiente llamada a `reduce`:

```
reduce(add, (1, 2, 3, 4))
```

realiza y devuelve el resultado del siguiente cálculo:

```
add(add(add(1, 2), 3), 4)
```

lo que, en la práctica, equivale a:

```
(((1 + 2) + 3) + 4)
```

Pero si el iterable es **vacío**, dará un error:

```
>>> reduce(add, ())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reduce() of empty iterable with no initial value
```

Con lo que acabamos de ver, se demuestra que la implementación de la función `reduce` en Python va reduciendo **de izquierda a derecha** y que, por tanto, las operaciones se hacen agrupándose **por la izquierda**.

Esto es algo que debemos tener muy en cuenta a la hora de diseñar la función que se le pasa a `reduce`.

Se denomina **iteración** a cada paso que da la función `reduce`, es decir, cada vez que `reduce` visita un nuevo elemento del iterable (la tupla, cadena o lo que sea) y aplica la función para calcular el resultado parcial.

Esa función, como ya dijimos antes, debe tener dos parámetros, pero de forma que, en cada iteración:

1. Su primer parámetro va a contener siempre el valor parcial acumulado hasta ahora (por tanto, es un *acumulador*).
2. Su segundo parámetro va a contener el valor del elemento que en este momento está visitando `reduce`.

Por tanto, es frecuente que el primer parámetro de esa función se llame `acc` o algo similar, para expresar el hecho de que ahí se va recibiendo el valor acumulado hasta el momento.

Por ejemplo, en la siguiente llamada:

```
reduce(lambda acc, e: acc + e, (1, 2, 3, 4), 0)
```

- `acc` va a contener la suma parcial acumulada hasta ahora.
- `e` va a contener el elemento que en este momento se está visitando.

Así, durante la ejecución del `reduce`, ésta provocará las siguientes llamadas a la expresión lambda:

```
(lambda acc, e: acc + e)(0, 1)   # acc = 0, e = 1
(lambda acc, e: acc + e)(1, 2)   # acc = 1, e = 2
(lambda acc, e: acc + e)(3, 3)   # acc = 3, e = 3
(lambda acc, e: acc + e)(6, 4)   # acc = 6, e = 4
```

¿Cómo podríamos definir la función `reduce` si recibiera una tupla y no cualquier iterable?

Ejercicio

4. Inténtalo.

Una forma (con valor inicial obligatorio) podría ser así:

```
def reduce(fun, tupla, ini):
    if len(tupla) == 0:
        return ini
    return reduce(fun, tupla[1:], fun(ini, tupla[0]))
```

O bien, de forma iterativa y no recursiva:

```
def reduce(fun, tupla, ini):
    res = ini
    for e in tupla:
        res = fun(res, e)
    return res
```

3.5. Expresiones generadoras

Dos operaciones que se realizan con frecuencia sobre un iterable son:

- Realizar alguna operación sobre cada elemento (`map`).
- Seleccionar un subconjunto de elementos que cumplan alguna condición (`filter`).

Las **expresiones generadoras** son una notación copiada del lenguaje Haskell que nos permite realizar ambas operaciones de una forma muy concisa.

El resultado que devuelve es un iterador.

Por ejemplo:

```
>>> tuple(x ** 3 for x in (1, 2, 3, 4))
(1, 8, 27, 64)
# equivale a:
>>> tuple(map(lambda x: x ** 3, (1, 2, 3, 4)))
(1, 8, 27, 64)
```

```
>>> tuple(x for x in (-4, 3, 5, -2, 8, -3, 9) if x > 0)
(3, 5, 8, 9)
# equivale a:
>>> tuple(filter(lambda x: x > 0, (-4, 3, 5, -2, 8, -3, 9)))
(3, 5, 8, 9)
```

Su sintaxis es:

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

Los elementos de la salida generada serán los sucesivos valores de ⟨*expresión*⟩.

Las cláusulas **`if`** son opcionales. Si están, la ⟨*expresión*⟩ sólo se evaluará y añadirá al resultado cuando se cumpla la ⟨*condición*⟩.

Los paréntesis `(` y `)` alrededor de la expresión generadora se pueden quitar si la expresión se usa como único argumento de una función.

Por ejemplo:

```
>>> sec1 = 'abc'
>>> sec2 = (1, 2, 3)
>>> tuple((x, y) for x in sec1 for y in sec2)
(('a', 1), ('a', 2), ('a', 3),
```

```
 ('b', 1), ('b', 2), ('b', 3),
 ('c', 1), ('c', 2), ('c', 3))
```

Las expresiones generadoras, al igual que las expresiones lambda, **determinan su propio *ámbito***.

Ese ámbito abarca toda la expresión generadora, de principio a fin.

Los identificadores que aparecen en la cláusula **`for`** se se van ligando automáticamente, uno a uno, a cada elemento del iterable indicada en la cláusula **`in`**.

Al recorrer el iterable, las variables van almacenando en cada iteración del bucle el valor del elemento que en ese momento se está visitando.

Debido a ello, podemos afirmar que las variables que aparecen en en cada cláusula `for` de la expresión generadora son **identificadores cuantificados**, ya que toman sus valores automáticamente y éstos están restringido a los valores que devuelva el iterable.

Además, estos identificadores cuantificados son locales a la expresión generadora, y sólo existen dentro de ella.

Debido a lo anterior, esos identificadores cumplen estas dos propiedades:

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

  Por ejemplo, las dos expresiones generadoras 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 expresión generadora, ya que estarían fuera de su ámbito y no serían visibles.

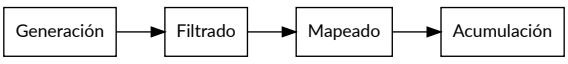
  Por ejemplo, lo siguiente daría un error de nombre:

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

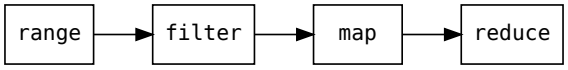
3.6. Procesamiento de flujos

El **procesamiento de flujos** es una técnica de diseño de programas que se basa en estudiar cómo se transforman los datos de entrada en datos de salida y descomponer dicha transformación en una secuencia de etapas intermedias donde la salida de una etapa es la entrada de la siguiente, como en una cadena de montaje.

El esquema general es el siguiente, con posibles ligeras variaciones:



Este esquema se puede trasladar al uso de funciones ya conocidas, de la siguiente forma:

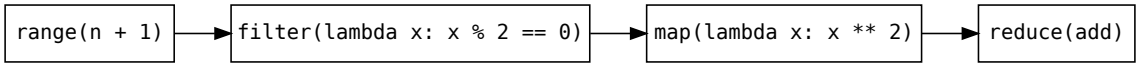


Procesamiento de flujos con funciones

Por ejemplo, supongamos el siguiente problema:

«*Dada una lista con los n primeros números naturales, obtener la suma de los cuadrados de los números impares que haya en la lista.*»

Este problema se podría resolver de la siguiente forma:



Solución mediante procesamiento de flujos

4. Funciones genéricas

4.1. Definición y uso

Esquemáticamente, las **funciones genéricas** tienen la siguiente forma:

```
def func[T](arg: T): ...
```

En esta definición, `T` es una **variable de tipo**, es decir, un identificador que representa a un *tipo* cualquiera que en este momento no está determinado.

Al usar la sintaxis `[T]`, decimos que `T` representa un **parámetro de tipo** para la función, y sirve para expresar el hecho de que la función que estamos definiendo es genérica y funciona con valores de muchos tipos distintos (uno por cada posible valor de `T`).

Esta forma de definir funciones caracteriza un cierto tipo de polimorfismo llamado **polimorfismo paramétrico**, donde una misma función puede actuar sobre valores de tipos muy diversos. Por eso, a esas funciones las podemos llamar **funciones polimórficas**.

Por ejemplo, la función que devuelve el máximo elemento de una lista donde todos sus elementos son del mismo tipo, se puede anotar de la siguiente forma:

```
def maximo[T](l: list[T]) -> T:
    ...
```

y podríamos llamarla pasándole cualquier tipo de lista, siempre que todos los elementos de la lista sean del mismo tipo:

```
>>> maximo([1, 2, 3, 4])
4
```

```
>>> maximo(['a', 'b', 'c', 'd'])
'd'
```

Según la signatura de la función, no sería correcto pasarle una lista con elementos de diferentes tipos, pero el intérprete no detectaría el error ya que no hace comprobación de tipos. De todas formas, la función probablemente no funcionaría bien en ese caso, ya que es una violación de su especificación:

```
>>> maximo([1, 'a', True, 2.5])  # Incorrecto si se comprueban los tipos
???                              # No se sabe cómo se comportará la función en este caso
```

Una función genérica puede tener **más de un parámetro de tipo** en caso necesario.

Por ejemplo:

```
def crear_tupla[T, U](elem1: T, elem2: U) -> tuple[T, U]:
    """Crea una tupla con dos elementos."""
    return (elem1, elem2)
```

Un ejemplo de uso:

```
>>> crear_tupla(1, 'a')
(1, 'a')
```

Aquí, mandamos a la función un elemento de tipo `int` y otro de tipo `str`, y la función produce como resultado una tupla de tipo `tuple[int, str]`.

Bibliografía

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