Abstracciones funcionales

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/10/28 a las 23:38:00

Índice

1. Abstracciones lambda

1.1. Expresiones lambda

Las **expresiones lambda** (también llamadas **abstracciones lambda** o **funciones anónimas** en algunos lenguajes) son expresiones que capturan la idea abstracta de «**función**».

Son la forma más simple y primitiva de describir funciones en un lenguaje funcional.

Su sintaxis (simplificada) es:

⟨*expresión_lambda*⟩ ::= **`lambda`** [⟨*lista_parámetros*⟩]**`:`** ⟨*expresión*⟩
⟨*lista_parámetros*⟩ := **`identificador`** (**`,`** **`identificador`**)*

Por ejemplo, la siguiente expresión lambda captura la idea general de «*suma*»:

```
lambda x, y: x + y
```

1.2. Parámetros y cuerpos

Los identificadores que aparecen entre la palabra clave **`lambda`** y el carácter de dos puntos (`:`) son los **parámetros** de la expresión lambda.

La expresión que aparece tras los dos puntos (`:`) es el **cuerpo** de la expresión lambda.

En el ejemplo anterior:

```
lambda x, y: x + y
```

- Los parámetros son `x` e `y`.
- El cuerpo es `x + y`.
- Esta expresión lambda captura la idea general de sumar dos valores (que en principio pueden ser de cualquier tipo, siempre y cuando admitan el operador `+`).
- En sí misma, esa expresión devuelve un valor válido que representa a una función.

1.3. Aplicación funcional

De la misma manera que podemos aplicar una función a unos argumentos, también podemos aplicar una expresión lambda a unos argumentos.

Por ejemplo, la aplicación de la función `max` sobre los argumentos `3` y `5` es una expresión que se escribe como `max(3, 5)` y que denota el valor **cinco**.

Igualmente, la aplicación de una expresión lambda como

```
lambda x, y: x + y
```

sobre los argumentos `4` y `3` se representa así:

```
(lambda x, y: x + y)(4, 3)
```

O sea, que la expresión lambda representa el papel de una función.

1.3.1. Evaluación de una aplicación funcional

En nuestro *modelo de sustitución*, la **evaluación de la aplicación de una expresión lambda** consiste en **sustituir**, en el cuerpo de la expresión lambda, **cada parámetro por su argumento correspondiente** (por orden) y devolver la expresión resultante *parentizada* (o sea, entre paréntesis).

A esta operación se la denomina **aplicación funcional** o **β-reducción**.

Siguiendo con el ejemplo anterior:

```
(lambda x, y: x + y)(4, 3)
```

sustituimos en el cuerpo de la expresión lambda los parámetros `x` e `y` por los argumentos `4` y `3`, respectivamente, y parentizamos la expresión resultante, lo que da:

```
(4 + 3)
```

que simplificando (según las reglas del operador `+`) da `7`.

Es importante hacer notar que el cuerpo de una expresión lambda sólo se evalúa cuando se lleva a cabo una β-reducción (es decir, cuando se aplica la expresión lambda a unos argumentos), y no antes.

Por tanto, el cuerpo de la expresión lambda no se evalúa cuando se define la expresión.

Por ejemplo, al evaluar la expresión:

```
lambda x, y: x + y
```

el intérprete no evalúa la expresión del cuerpo (`x + y`), sino que crea un valor de tipo «función» pero sin entrar a ver «qué hay» en el cuerpo.

Sólo se mira lo que hay en el cuerpo cuando se aplica la expresión lambda a unos argumentos.

En particular, podemos tener una expresión lambda como la siguiente, que sólo dará error cuando se aplique a un argumento, no antes:

```
lambda x: x + 1/0
```

1.3.2. Funciones con nombre

Si hacemos la siguiente definición:

```
suma = lambda x, y: x + y
```

le estamos dando un nombre a la expresión lambda, es decir, a una función.

A partir de ese momento podemos usar `suma` en lugar de su valor (la expresión lambda), por lo que podemos hacer:

```
suma(4, 3)
```

en lugar de

```
(lambda x, y: x + y)(4, 3)
```

Cuando aplicamos a sus argumentos una función así definida también podemos decir que estamos **invocando** o **llamando** a la función. Por ejemplo, en `suma(4, 3)` estamos *llamando* a la función `suma`, o hay una *llamada* a la función `suma`.

La evaluación de la llamada a `suma(4, 3)` implicará realizar los siguientes tres pasos y en este orden:

1. Sustituir el nombre de la función `suma` por su definición, es decir, por la expresión lambda a la cual está ligado.
2. Evaluar los argumentos que aparecen en la llamada.
3. Aplicar la expresión lambda a sus argumentos (β-reducción).

Esto implica la siguiente secuencia de reescrituras:

```
suma(4, 3)                    # evalúa suma y devuelve su definición
= (lambda x, y: x + y)(4, 3)  # evalúa 4 y devuelve 4
= (lambda x, y: x + y)(4, 3)  # evalúa 3 y devuelve 3
= (lambda x, y: x + y)(4, 3)  # aplica la expresión lambda sus argumentos
= (4 + 3)                     # evalúa 4 + 3 y devuelve 7
= 7
```

Como una expresión lambda es una función, **aplicar una expresión lambda a unos argumentos es como llamar a una función pasándole dichos argumentos**.

Por tanto, también podemos decir que ***llamamos* o *invocamos* una expresión lambda**, pasándole unos argumentos durante esa llamada.

En consecuencia, ampliamos ahora nuestra gramática de las expresiones en Python incorporando las expresiones lambda como un tipo de función:

⟨*llamada_función*⟩ ::= ⟨*función*⟩**(**[⟨*lista_argumentos*⟩]**)**
⟨*función*⟩ ::= **identificador**
| **(**⟨*expresión_lambda*⟩**)**
⟨*expresión_lambda*⟩ ::= **lambda** [⟨*lista_parámetros*⟩]**:** ⟨*expresión*⟩
⟨*lista_parámetros*⟩ ::= **identificador**(**,** **identificador**)*
⟨*lista_argumentos*⟩ ::= ⟨*expresión*⟩(**,** ⟨*expresión*⟩)*

Ejemplo

Dado el siguiente código:

```
suma = lambda x, y: x + y
```

¿Cuánto vale la expresión siguiente?

```
suma(4, 3) * suma(2, 7)
```

Según el modelo de sustitución, reescribimos:

```
suma(4, 3) * suma(2, 7)                    # definición de suma
= (lambda x, y: x + y)(4, 3) * suma(2, 7)  # evaluación de 4
= (lambda x, y: x + y)(4, 3) * suma(2, 7)  # evaluación de 3
= (lambda x, y: x + y)(4, 3) * suma(2, 7)  # aplicación a 4 y 3
= (4 + 3) * suma(2, 7)                     # evaluación de 4 + 3
= 7 * suma(2, 7)                           # definición de suma
= 7 * (lambda x, y: x + y)(2, 7)           # evaluación de 2
= 7 * (lambda x, y: x + y)(2, 7)           # evaluación de 7
= 7 * (lambda x, y: x + y)(2, 7)           # aplicación a 2 y 7
= 7 * (2 + 7)                              # evaluación de 2 + 7
= 7 * 9                                    # evaluación de 7 * 9
= 63
```

1.3.3. Composición de funciones

Podemos crear una función que use otra función. Por ejemplo, para calcular el área de un círculo usamos otra función que calcule el cuadrado de un número:

```
cuadrado = lambda x: x * x
area = lambda r: 3.1416 * cuadrado(r)
```

La expresión `area(11 + 1)` se evaluaría así:

```
1 area(11 + 1)                                # definición de area
2 = (lambda r: 3.1416 * cuadrado(r))(11 + 1)  # evalúa 11 y devuelve 11
3 = (lambda r: 3.1416 * cuadrado(r))(11 + 1)  # evalúa 1 y devuelve 1
4 = (lambda r: 3.1416 * cuadrado(r))(11 + 1)  # evalúa 11 + 1 y devuelve 12
```

```
= (lambda r: 3.1416 * cuadrado(r))(12)      # aplicación a 12
= (3.1416 * cuadrado(12))                   # evalúa 3.1416 y devuelve 3.1416
= (3.1416 * cuadrado(12))                   # definición de cuadrado
= (3.1416 * (lambda x: x * x)(12))          # aplicación a 12
= (3.1416 * (12 * 12))                      # evalúa (12 * 12) y devuelve 144
= (3.1416 * 144)                            # evalúa (3.1416 * 11) y...
= 452.3904                                  # ... devuelve 452.3904
```

En detalle:

- **Línea 1**: Se evalúa `area`, que devuelve su definición (una expresión lambda).
- **Líneas 2–4**: Lo siguiente a evaluar es la aplicación de la expresión lambda de `area` sobre su argumento, por lo que primero evaluamos éste.
- **Línea 5**: Ahora se aplica la expresión lambda a su argumento `12`.
- **Línea 6**: Lo siguiente que toca evaluar es el `3.1416`, que ya está evaluado.
- **Línea 7**: A continuación hay que evaluar la aplicación de `cuadrado` sobre `12`. Primero se evalúa `cuadrado`, sustituyéndose por su definición...
- **Línea 8**: ... y ahora se aplica la expresión lambda a su argumento `12`.
- Lo que queda ya por evaluar es todo aritmética.

A veces no resulta fácil determinar el orden en el que hay que evaluar las subexpresiones que forman una expresión, sobre todo cuando se mezclan funciones y operadores en una misma expresión.

En ese caso, puede resultar útil reescribir los operadores como funciones, cuando sea posible, y luego dibujar el árbol sintático correspondiente a esa expresión, para ver a qué profundidad quedan los nodos.

Por ejemplo, la siguiente expresión:

```
abs(-12) + max(13, 28)
```

se puede reescribir como:

```
from operator import add
add(abs(-12), max(13, 28))
```

y si dibujáramos el árbol sintáctico veríamos que la suma está más arriba que el valor absoluto y el máximo (que están, a su vez, al mismo nivel de profundidad).

Un ejemplo más complicado:

```
abs(-12) * max((2 + 3) ** 5), 37)
```

se reescribiría como:

```
from operator import add, mul
mul(abs(-12), max(pow(add(2, 3), 5), 37))
```

donde se aprecia claramente que el orden de las operaciones, de más interna a más externa, sería:

1. Suma (`+` o `add`).
2. Potencia (`**` o `pow`).
3. Valor absoluto (`abs`) y máximo (`max`) al mismo nivel.
4. Producto (`*` o `mul`).

1.4. Identificadores cuantificados y libres de una expresión lambda

Si un *identificador* de los que aparecen en el *cuerpo* de una expresión lambda también aparece en la *lista de parámetros* de esa expresión lambda, decimos que es un **identificador cuantificado** de la expresión lambda.

En caso contrario, le llamamos **identificador libre** de la expresión lambda.

En el ejemplo anterior:

```
lambda x, y: x + y
```

los dos identificadores que aparecen en el cuerpo (`x` e `y`) aparecen también en la lista de parámetros de la expresión lambda, por lo que ambos son identificadores cuantificados y no hay ningún identificador libre.

En cambio, en la expresión lambda:

```
lambda x, y: x + y + z
```

`x` e `y` son identificadores cuantificados (porque aparecen en la lista de parámetros de la expresión lambda), mientras que `z` es un identificador libre.

En realidad, **un *identificador cuantificado* y un *parámetro* están vinculados, hasta el punto en que podemos considerar que son la misma cosa**.

Tan sólo cambia su denominación dependiendo del lugar donde aparece su identificador en la expresión lambda:

- Cuando aparece **antes** del «`:`», le llamamos «*parámetro*».
- Cuando aparece **después** del «`:`», le llamamos «*identificador cuantificado*».

Por ejemplo: en la siguiente expresión lambda:

```
lambda x, y: x + y
       ┬     ┬
       │     └──────── identificador cuantificado
       └── parámetro
```

el identificador `x` aparece dos veces, pero en los dos casos representa la misma cosa. Tan sólo se llama de distinta forma («*parámetro*» o «*identificador cuantificado*») dependiendo de dónde aparece.

A los identificadores cuantificados se les llama así porque sus posibles valores están cuantificados o *restringidos* a los posibles valores que puedan tomar los parámetros de la expresión lambda en cada llamada a la misma.

Dicho valor además vendrá determinado automáticamente por la ligadura que crea el intérprete durante la llamada a la expresión lambda.

Es decir: el intérprete liga automáticamente el identificador cuantificado al valor del correspondiente argumento durante la llamada a la expresión lambda.

En cambio, el valor al que esté ligado un identificador libre de una expresión lambda no viene determinado por ninguna característica propia de dicha expresión lambda.

2. Abstracciones funcionales

2.1. Encapsulación y cajas negras

Encapsular es encerrar varios elementos juntos dentro una ***cápsula*** que se puede manipular como una sola unidad, de forma que parte de lo que hay dentro queda visible y accesible desde el exterior, mientras que el resto queda oculto e inaccesible en el interior.

La **encapsulación** es el mecanismo que proporcionan los lenguajes de programación para que el programador pueda encapsular elementos de un programa, y puede verse al mismo tiempo como un mecanismo de *agrupamiento* y como un mecanismo de de *protección*.

La *membrana* de la cápsula separa el exterior del interior de la misma, y es una membrana *permeable* porque permite exponer al exterior ciertos elementos que así resultan visibles y accesibles desde fuera de ella.

Esa membrana permite también la entrada y salida de señales y datos desde y hacia el exterior de la cápsula.

La parte que la membrana de la cápsula expone al exterior de la misma se denomina la **interfaz** de la cápsula, e incluye también los puntos de entrada y salida de señales y datos antes mencionados.

En general, la **interfaz** de una cápsula es todo aquello que nos permite interactuar con la cápsula desde el exterior de la misma y, en sentido contrario, también representa la forma que tiene la cápsula de mandar señales o datos al exterior.

La cápsula tiene un espacio de nombres local que guarda las ligaduras que van dentro de la cápsula y que previenen el *name clash*, haciendo que una cápsula pueda contener elementos internos con el mismo nombre que otros elementos externos a la misma sin que haya conflictos.

Una **caja negra** es una cápsula que:

- *expone* justamente aquello que es necesario conocer para poder usarla (el ***qué*** hace), y
- *oculta* todos los demás detalles internos de funcionamiento (el ***cómo*** lo hace).

La «tapa» de la caja negra (lo que incluye su *interfaz*) es precisamente la barrera de separación entre el qué hace y el cómo lo hace.

Con una caja negra sólo se puede interactuar mediante sus entradas y salidas, sin necesidad (ni posibilidad) de saber lo que hay dentro.

Abstraer es quedarse con la idea esencial de aquello que se está estudiando. El producto resultante de ese proceso se llama ***abstracción***.

Por tanto, la **abstracción** es, al mismo tiempo, una acción y un producto:

- Como ***acción***, es el proceso mental que consiste en centrarse en lo que es importante y esencial en un determinado momento e ignorar los detalles que en ese momento no resultan importantes.
- Como ***producto***, es una cápsula que contiene un mecanismo más o menos complejo y a la que se le da un nombre. De esta forma, podemos referirnos a todo el mecanismo simplemente usando ese nombre sin tener que conocer su composición interna ni sus detalles internos de funcionamiento.

Por tanto, para usar la abstracción nos bastará con conocer su *nombre* y saber *qué hace*, sin necesidad de saber *cómo lo hace* ni qué elementos la forman *internamente*.

En consecuencia, las abstracciones están *encapsuladas*, pero no de cualquier manera, sino de forma que:

- lo que queda visible desde el exterior de la cápsula es *qué hace la abstracción*, y
- lo que se oculta en el interior es *cómo lo hace*.

Eso quiere decir que **las abstracciones se pueden representar como *cajas negras*** o, dicho de otra manera, que **las cajas negras son una forma de representar abstracciones**.

Esas abstracciones se denominan **abstracciones de caja negra** (*black box abstractions*), y son las que trabajaremos en este curso.

Creamos abstracciones para:

- Controlar la complejidad.
- Expresar conceptos abstractos.
- Evitar repeticiones de código.
- Crear casos generales a partir de patrones que se repiten.

2.2. Las funciones como abstracciones

Recordemos la definición de la función `area`:

```
cuadrado = lambda x: x * x
area = lambda r: 3.1416 * cuadrado(r)
```

Aunque es muy sencilla, la función `area` ejemplifica la propiedad más potente de las funciones definidas por el programador: la **abstracción**.

La función `area` está definida sobre la función `cuadrado`, pero sólo necesita saber de ella qué resultados de salida devuelve a partir de sus argumentos de entrada (o sea, ***qué*** calcula y no ***cómo*** lo calcula).

Podemos escribir la función `area` sin preocuparnos de cómo calcular el cuadrado de un número, porque eso ya lo hace la función `cuadrado`.

Los detalles sobre cómo se calcula el cuadrado están **ocultos dentro de la definición** de `cuadrado`. Esos detalles **se ignoran en este momento** al diseñar `area`, para considerarlos más tarde si hiciera falta.

De hecho, por lo que respecta a `area`, `cuadrado` no representa una definición concreta de función, sino más bien la abstracción de una función, lo que se denomina una **abstracción funcional**, ya que a `area` le sirve igual de bien cualquier función que calcule el cuadrado de un número.

Por tanto, si consideramos únicamente los valores que devuelven, las tres funciones siguientes son indistinguibles e igual de válidas para `area`. Ambas reciben un argumento numérico y devuelven el cuadrado de ese número:

```
cuadrado = lambda x: x * x
cuadrado = lambda x: x ** 2
cuadrado = lambda x: x * (x - 1) + x
```

En otras palabras: la definición de una función debe ser capaz de **ocultar sus detalles internos de funcionamiento**, ya que para usar la función no debe ser necesario conocer esos detalles.

2.2.1. Control de la complejidad

La abstracción es el principal mecanismo de control de la complejidad, ya que nos permite ocultar detrás de un nombre los detalles que componen una parte del programa, haciendo que esa parte actúe (a ojos del programador que la utilice) como si fuera un elemento *predefinido* del lenguaje, de forma que el programador lo puede usar sin tener que saber cómo funciona por dentro.

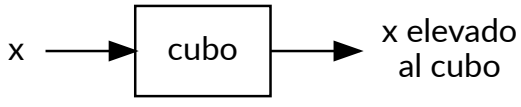
Desde ese punto de vista, podemos decir que **las funciones son abstracciones** porque nos permiten usarlas sin tener que conocer los detalles internos del procesamiento que realizan.

Por ejemplo, si queremos usar la función `cubo` (que calcula el cubo de un número), nos da igual que dicha función esté implementada de cualquiera de las siguientes maneras:

```
cubo = lambda x: x * x * x
cubo = lambda x: x ** 3
cubo = lambda x: x * x ** 2
```

Para **usar** la función, nos basta con saber que calcula el cubo de un número, sin necesidad de saber qué operaciones concretas realiza para obtener el resultado.

Los detalles de implementación quedan ocultos y por eso también decimos que `cubo` es una **abstracción**, la cual se puede representar con una **caja negra** con una entrada y una salida:



La función `cubo` como caja negra

2.2.2. Definición de conceptos abstractos

Las funciones también son abstracciones porque nos permiten definir **conceptos abstractos**.

Por ejemplo, cuando definimos:

```
cubo = lambda x: x * x * x
```

estamos definiendo en qué consiste *elevar «algo» al cubo*, es decir, estamos creando un concepto que antes no existía, y se lo estamos enseñando a nuestro lenguaje.

De esta forma, nuestro lenguaje ya sabrá qué es elevar algo al cubo, y podremos usarlo en nuestros programas.

Por supuesto, nos las podemos arreglar sin definir el concepto de *cubo*, escribiendo siempre expresiones explícitas (como `3*3*3`, `5*5*5`, etc.) sin usar la palabra «`cubo`», pero eso nos obligaría siempre a expresarnos usando las operaciones primitivas de nuestro lenguaje (como `*`), en vez de poder usar términos de más alto nivel.

Es decir: **nuestros programas podrían calcular el cubo de un número, pero no tendrían la habilidad de expresar el concepto de *elevar al cubo***.

2.2.3. Repetición de código

Las funciones (y, en general, cualquier abstracción) nos permiten evitar la aparición de código repetido en nuestros programas.

Cuando encontramos el mismo código (exactamente el mismo) repetido en varios puntos diferentes del mismo programa, es conveniente crear una abstracción que encapsule ese código repetido y le dé un nombre.

Así, podremos usar el nombre de la abstracción en lugar del código repetido en aquellas partes del programa donde se encuentre éste.

2.2.4. Generalización

Las funciones también son **generalizaciones** de casos particulares porque describen operaciones compuestas a realizar sobre ciertos valores sin importar cuáles sean esos valores en concreto.

Por ejemplo, cuando definimos:

```
cubo = lambda x: x * x * x
```

no estamos hablando del cubo de un número en particular, sino más bien de un **método** para calcular el cubo de cualquier número.

De esta forma, podemos usar la función para obtener los diferentes casos particulares que obtendríamos si no tuviéramos la función.

Por ejemplo, podremos usar `cubo(3)` en lugar de `3*3*3`, `cubo(5)` en lugar de `5*5*5`, etc.

Es decir, estamos expresando cómo se calcula el cubo de cualquier número, en general.

Una de las habilidades que deberíamos pedir a un lenguaje potente es la posibilidad de **construir abstracciones** asignando nombres a los patrones más comunes, y luego trabajar directamente usando dichas abstracciones.

Las funciones nos permiten esta habilidad, y esa es la razón de que todos los lenguajes (salvo los más primitivos) incluyan mecanismos para definir funciones.

Por ejemplo: en el caso anterior, vemos que hay un patrón (multiplicar algo por sí mismo tres veces) que se repite con frecuencia, y a partir de él construimos una abstracción que asigna un nombre a ese patrón (*elevar al cubo*).

Esa abstracción la definimos como una función que describe la *regla* necesaria para elevar algo al cubo.

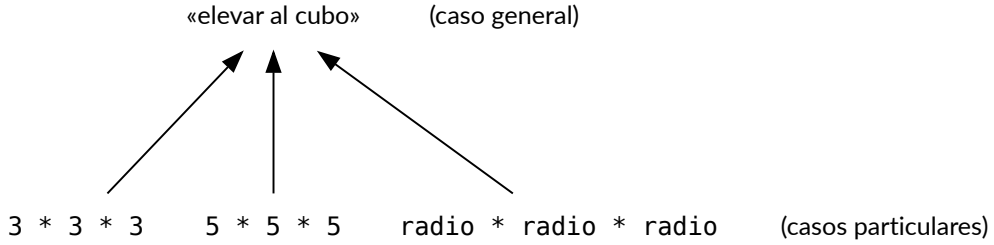
Esa técnica combina la abstracción con la generalización.

De esta forma, analizando ciertos *casos particulares*, observamos que se repite el mismo patrón en todos ellos (es decir, **abstraemos** el concepto esencial), y de ahí extraemos un *caso general* (es decir, hacemos una **generalización**) que agrupa a todos los posibles casos particulares que cumplen ese patrón.

Luego, hacemos una **encapsulación**, metiendo ese caso general en una «*caja negra*» que oculte sus detalles internos, y finalmente le damos un nombre a la «caja», con lo que acabamos creando una **abstracción**.

Por ejemplo, cuando vemos que en nuestros programas es frecuente tener que multiplicar una cosa por sí misma tres veces, deducimos que ahí hay un patrón común que se repite en todos los casos.

De ahí, creamos la abstracción que describe ese patrón general y le llamamos «*elevar al cubo*»:



Ese patrón general representa a cada miembro del grupo formado por sus casos particulares, y se construye colocando un *parámetro* allí donde los casos particulares se diferencian entre sí (es decir, hacemos una **parametrización**).

La función resultante es, al mismo tiempo:

- una **encapsulación** (porque los detalles internos de la función quedan ocultos dentro del cuerpo de la expresión lambda como si fuera una caja negra),
- una **abstracción** (porque se puede invocar a la función usando simplemente su nombre sin necesidad de saber cómo está hecha por dentro y, por tanto, sabiendo *qué* sin tener que saber *cómo* lo hace), y
- una **generalización** (porque, al estar parametrizada, representa muchos casos particulares con un único caso general).

Al invocar a la función, se ligan sus parámetros con los argumentos de la llamada, lo que produce un caso particular a partir del caso general.

En resumen, el proceso conceptual sería el siguiente:

1. Observar que hay varios casos particulares que se parecen según un patrón común y que representan el mismo concepto.
2. Generalizar esos casos particulares y parametrizar el caso general creando una expresión lambda.
3. Abstraer la expresión lambda dándole un nombre.

Veamos cada paso por separado con detalle.

Paso 1: Abstracción

Partimos de casos particulares que se parecen. Por ejemplo, supongamos las siguientes expresiones:

```
3 * 3 * 3
5 * 5 * 5
```

Si las comparamos, vemos que tienen la misma forma y que contienen elementos que son iguales y otros que son diferentes.

Por ejemplo, los operadores `*` son iguales, y lo que varía es la cosa que se multiplica (el `3`, el `5`, etcétera).

A partir de ahí, hacemos abstracción y nos centramos en estudiar aquello en lo que se parecen e ignoramos aquello en lo que se diferencian.

Haciendo eso, deducimos un patrón común que subyace a todas esas expresiones.

En este caso, el patrón es que en todas ellas hay «algo» que se multiplica por sí mismo tres veces.

Al deducir ese patrón estamos realizando un proceso de abstracción, porque nos estamos centrando en lo que ahora mismo importa (es un producto de «cosas») e ignorando los detalles que ahora mismo no importan (qué son esas «cosas»).

Ese patrón representa el concepto abstracto de «elevar al cubo».

Ahora bien: ¿cómo se materializa ese concepto?

Paso 2: Generalización

Generalizamos estos casos particulares, *parametrizando* los elementos en los que se diferencian (es decir, las partes que no son comunes) y dejando el resto igual.

En nuestro ejemplo, los casos particulares se distinguen en el número que se multiplica para calcular el cubo. Por tanto, habrá un único parámetro que representará dicho número.

Incorporamos el parámetro en la expresión sustituyendo, en uno cualquiera de los casos particulares, el número que se multiplica por el nombre que escojamos para el parámetro (el cual debe ser un identificador que no estuviera ya en la expresión) y utilizamos el cuantificador **`lambda`**, creando así una ***expresión lambda***:

```
lambda x: x * x * x
```

Los **parámetros** son nombres que tomarán los valores de los argumentos aplicados a la expresión lambda. En este caso, la `x` es el único parámetro que tiene la expresión lambda anterior.

Esa expresión describe el patrón común como un **caso general** de las expresiones anteriores, ya que representa a todos los posibles casos particulares (potencialmente infinitos) que se ajustan a ese mismo patrón. Por ese motivo decimos que es una **generalización**.

Un ***cuantificador*** es un símbolo que convirte constantes en variables. En este caso, convierte las constantes `3`, `5`, etc. en el identificador `x`.

Ese identificador está cuantificado porque está afectado por el cuantificador `lambda`. De lo contrario, sería un identificador *libre*.

Al invocarla con un argumento concreto, el parámetro toma el valor de ese argumento y así se van obteniendo los casos particulares deseados:

```
(lambda x: x * x * x)(3) → 3 * 3 * 3
(lambda x: x * x * x)(5) → 5 * 5 * 5
```

La expresión lambda puede usarse simplemente aplicando los argumentos necesarios en cada llamada, sin necesidad de manipular directamente la expresión que forma su cuerpo y que es la que lleva a cabo el procesamiento y el cálculo del resultado.

Por eso podemos decir que la expresión lambda está *encapsulada* formando una **caja negra** con una parte expuesta, visible y manipulable desde el exterior (sus parámetros) y otra parte oculta dentro de la cápsula (su cuerpo), que no necesitaríamos manipular para poder usar la expresión lambda.

Paso 3: Más abstracción

Aunque para usar una expresión lambda no necesitamos conocer cómo es su cuerpo, en la práctica sí que podemos verlo (aunque no podemos manipularlo directamente):

```
(lambda x: x * x * x)(3) → 3 * 3 * 3
```

Una expresión lambda sin nombre es como una función de «usar y tirar» que vive y muere en la misma expresión donde se la utiliza.

En general, las **abstracciones de usar y tirar** son aquellas que se crean para cumplir una función concreta, en un punto específico del programa, y después no se reutilizan ni se les da un nombre permanente.

Son abstracciones efímeras, que encapsulan algo pero sin la intención de volver a usarlas más adelante.

En cambio, cuando le damos un nombre, subimos más el nivel de abstracción ya que puede reutilizarse en muchas expresiones y toda la expresión lambda queda «oculta» bajo el nombre que le damos.

Así, ahora podemos usar su nombre en lugar de la expresión lambda y con ello creamos una **abstracción lambda**:

```
cubo = lambda x: x * x * x
```

De ahora en adelante, podemos invocar a la función usando su nombre, sin tener que recordar incluso que debajo hay una expresión lambda concreta:

```
cubo(3) → 3 * 3 * 3
cubo(5) → 5 * 5 * 5
```

La función `cubo` así creada **es una abstracción** porque:

- Para usarla sólo basta con saber su nombre y *qué* hace.
- No es necesario saber *cómo* lo hace.
- Es una caja negra que expone el *qué* y oculta el *cómo*.

Ahora ya ni siquiera tenemos que saber cómo es la expresión lambda a la que está ligada el nombre. Por tanto, trabajamos a un nivel mayor de abstracción.

La importancia de la **abstracción** reside en su capacidad para ocultar detalles irrelevantes y en el uso de nombres para referenciar objetos.

La principal preocupación del usuario de un programa (o de cada una de sus partes) es *qué* hace. Esto contrasta con la del programador de esa parte del programa, cuya principal preocupación es *cómo* lo hace.

Los lenguajes de programación proporcionan abstracción mediante funciones (y otros elementos como procedimientos y módulos, que veremos posteriormente) que permiten al programador distinguir entre lo que hace una parte del programa y cómo se implementa esa parte.

Una función, además, **encapsula** porque crea una *cápsula* alrededor de ella que sólo deja visible al exterior parte de su contenido; en particular, la cápsula de una función sólo deja pasar fuera lo necesario para poder usar la función, y oculta dentro todo lo demás, es decir, lo que no es necesario conocer ni manipular para usarla.

La abstracción es esencial en la construcción de programas. Pone el énfasis en lo que algo es o hace, más que en cómo se representa o cómo funciona. Por lo tanto, es el principal medio para gestionar la complejidad en programas grandes.

De igual importancia es la **generalización**.

Mientras que la abstracción reduce la complejidad al ocultar detalles irrelevantes, la generalización la reduce al sustituir, con una sola construcción, varios elementos que realizan una funcionalidad similar.

Los lenguajes de programación permiten la generalización mediante variables, parametrización, genéricos y polimorfismo.

La generalización es esencial en la construcción de programas. Pone el énfasis en las similitudes entre elementos. Por lo tanto, ayuda a gestionar la complejidad al agrupar individuos y proporcionar un representante que puede utilizarse para especificar cualquier individuo del grupo.

Resumen

Encapsulación: Es agrupar varios elementos juntos formando una sola unidad, ocultando algunos y exponiendo otros.

Cápsula: Es el resultado de la encapsulación. La membrana de la cápsula separa lo que se expone de lo que se oculta al exterior.

Caja negra: Es una cápsula que expone sólo lo necesario para poder usarla y oculta el resto de detalles innecesarios. La «tapa» de la caja negra separa el *qué* hace del *cómo* lo hace.

Abstracción: Conceptualmente, es el proceso de simplificar algo resaltando solo sus características esenciales y ocultando los detalles irrelevantes para el contexto en que se usa.

En la práctica, también es el producto resultante de ese proceso. En tal caso, una abstracción consiste en darle un nombre a una caja negra que expone la información necesaria para saber *qué* hace la abstracción y oculta los detalles necesarios para saber *cómo* lo hace.

Generalización: Es el proceso de identificar un patrón común entre varios casos particulares y crear un modelo más general que los abarque a todos.

Parametrización: Es el proceso de definir un elemento que representa un caso general utilizando *parámetros* que permiten obtener los casos particulares del caso general sin tener que reescribirlo, ligando valores concretos a los parámetros.

Parámetro: Es una parte que cambia en cada caso particular de un patrón común.

En resumen, creamos abstracciones:

- Cuando queremos **reducir la complejidad**, dándole un nombre a un mecanismo complejo para poder referirnos a todo el conjunto a través de su nombre sin tener que recordar continuamente qué piezas contiene el mecanismo o cómo funciona éste por dentro.
- Cuando queremos que nuestro programa pueda **expresar un concepto abstracto**, como el de «elevar al cubo».
- Cuando queremos evitar **repeticiones de código** en nuestro programa, de forma que el mismo concepto aparezca sólo una vez, evitando problemas de inconsistencia y mantenimiento del código.
- Cuando creamos **casos generales a partir de patrones que se repiten** en varios casos particulares.

2.3. Pureza

Si una expresión lambda no contiene identificadores libres, el valor que obtendremos al aplicarla a unos argumentos dependerá únicamente del valor que tengan esos argumentos (no dependerá de nada que sea «*exterior*» a la expresión lambda).

En cambio, si el cuerpo de una expresión lambda contiene identificadores libres, el valor que obtendremos al aplicarla a unos argumentos no sólo dependerá del valor de los argumentos, sino también de los valores a los que estén ligados esos identificadores libres en el momento de evaluar la aplicación de la expresión lambda.

Es el caso del siguiente ejemplo, donde tenemos una expresión lambda que contiene un identificador libre (`z`) y, por tanto, cuando la aplicamos a los argumentos `4` y `3` obtenemos un valor que depende no sólo de los valores de `x` e `y` sino también del valor de `z` fuera de la función:

```
>>> prueba = lambda x, y: x + y + z
>>> z = 9
>>> prueba(4, 3)
16
```

En este otro ejemplo, tenemos una expresión lambda que calcula la suma de tres números a partir de otra expresión lambda que calcula la suma de dos números:

```
suma = lambda x, y: x + y
suma3 = lambda x, y, z: suma(x, y) + z
```

En este caso, hay un identificador (`suma`) que no aparece en la lista de parámetros de la expresión lambda ligada a `suma3`.

En consecuencia, el valor de dicha expresión lambda dependerá de lo que valga `suma` fuera de `suma3`.

Se dice que una expresión lambda es **pura** si, siempre que la apliquemos a unos argumentos, el valor obtenido va a depender únicamente del valor de esos argumentos o, lo que es lo mismo, del valor de sus parámetros en la llamada.

Podemos decir que hay distintos **grados de pureza**:

- Una expresión lambda en cuyo cuerpo no hay ningún identificador libre es **más pura** que otra que contiene identificadores libres.
- Una expresión lambda cuyos **identificadores libres** representan **funciones** que se usan en el cuerpo de la expresión lambda, es **más pura** que otra cuyos identificadores libres representan cualquier otro tipo de valor.

En el ejemplo anterior, tenemos que la expresión lambda de `suma3`, sin ser *totalmente pura*, a efectos prácticos se la puede considerar **pura**, ya que su único identificador libre (`suma`) se usa como una **función**.

Por ejemplo, las siguientes expresiones lambda están ordenadas de mayor a menor pureza, siendo la primera totalmente **pura**:

```
# producto es una expresión lambda totalmente pura:
producto = lambda x, y: x * y
# cuadrado es casi pura; a efectos prácticos se la puede
# considerar pura ya que sus identificadores libres (en este
# caso, sólo una: producto) son funciones:
cuadrado = lambda x: producto(x, x)
# suma es impura, porque su identificador libre (z) no es una función:
suma = lambda x, y: x + y + z
```

La pureza de una función es un rasgo deseado y que hay que tratar de alcanzar siempre que sea posible, ya que facilita el desarrollo y mantenimiento de los programas, además de simplificar el razonamiento sobre los mismos, permitiendo aplicar directamente nuestro modelo de sustitución.

Es más incómodo trabajar con `suma` porque hay que *recordar* que depende de un valor que está *fuera* de la expresión lambda, cosa que no resulta evidente a no ser que mires en el cuerpo de la expresión lambda.

2.4. Especificaciones de funciones

La **especificación de una *función*** es la descripción de **qué** hace la función sin entrar a detallar **cómo** lo hace.

La **implementación de una *función*** es la descripción de **cómo** hace lo que hace, es decir, los detalles de su algoritmo interno.

Para poder usar una función, un programador no debe necesitar saber cómo está implementada.

Eso es lo que ocurre, por ejemplo, con las funciones predefinidas del lenguaje (como `max`, `abs` o `len`): sabemos *qué* hacen pero no necesitamos saber *cómo* lo hacen.

Incluso puede que el usuario de una función no sea el mismo que la ha escrito, sino que la puede haber recibido de otro programador como una «**caja negra**», que tiene unas entradas y una salida pero no se sabe cómo funciona por dentro.

Para poder **usar una abstracción funcional** *nos basta* con conocer su *especificación*, porque es la descripción de qué hace esa función.

Igualmente, para poder **implementar una abstracción funcional** *necesitamos* conocer su *especificación*, ya que necesitamos saber *qué tiene que hacer* la función antes de diseñar *cómo va a hacerlo*.

La especificación de una abstracción funcional describe tres características fundamentales de dicha función:

- El **dominio**: el conjunto de datos de entrada válidos.
- El **rango** o **codominio**: el conjunto de posibles valores que devuelve.
- El **propósito**: qué hace la función, es decir, la relación entre su entrada y su salida.

Hasta ahora, al especificar **programas**, hemos llamado «**entrada**» al dominio, y hemos agrupado el rango y el propósito en una sola propiedad que llamamos «**salida**».

Por ejemplo, cualquier función `cuadrado` que usemos para implementar `area` debe satisfacer esta especificación:

$$\begin{cases} \textbf{Entrada} : n \in \mathbb{R} \\ \texttt{cuadrado} \\ \textbf{Salida} : n^2 \end{cases}$$

La especificación **no concreta cómo** se debe llevar a cabo el propósito. Esos son **detalles de implementación** que se abstraen a este nivel.

Este esquema es el que hemos usado hasta ahora para especificar programas, y se podría seguir usando para especificar funciones, ya que éstas son consideradas ***subprogramas*** (porciones de código que forman parte de un programa y que siguen el esquema de «*entrada - proceso - salida*» como cualquier programa).

Pero para especificar funciones resulta más adecuado usar el siguiente esquema, al que llamaremos **especificación funcional**:

$$\begin{cases} \textbf{Pre}: & \texttt{True} \\ & \texttt{cuadrado(}n\texttt{: float) -> float} \\ \textbf{Post}: & \texttt{cuadrado(}n\texttt{)} = n^2 \end{cases}$$

«**Pre**» representa la **precondición**: la propiedad que debe cumplirse justo *en el momento* de llamar a la función.

«**Post**» representa la **postcondición**: la propiedad que debe cumplirse justo *después* de que la función haya terminado de ejecutarse.

Lo que hay en medio es la **signatura**: el nombre de la función, el nombre y tipo de sus parámetros y el tipo del valor de retorno.

La especificación se lee así: «***Si se llama a la función respetando su signatura y cumpliendo su precondición, la llamada termina cumpliendo su postcondición***».

En este caso, la **precondición** es `True`, que equivale a decir que cualquier condición de entrada es buena para usar la función.

Dicho de otra forma: no hace falta que se dé ninguna condición especial para usar la función. Siempre que la llamada respete la signatura de la función, el parámetro *n* puede tomar cualquier valor de tipo `float` y no hay ninguna restricción adicional.

Por otro lado, la **postcondición** dice que al llamar a la función `cuadrado` con el argumento *n* se debe devolver n^2.

Tanto la precondición como la postcondición son **predicados**, es decir, expresiones lógicas que se escriben usando el lenguaje de las matemáticas y la lógica.

La **signatura** se escribe usando la sintaxis del lenguaje de programación que se vaya a usar para implementar la función (Python, en este caso).

Recordemos la diferencia entre:

- **Dominio** y **conjunto origen** de una función.
- **Rango** (o **codominio**) y **conjunto imagen** de una función.

¿Cómo recoge la especificación esas cuatro características de la función?

- La **signatura** expresa el **conjunto origen** y el **conjunto imagen** de la función.
- El **dominio** viene determinado por los valores del conjunto origen que cumplen la **precondición**.
- El **codominio** viene determinado por los valores del conjunto imagen que cumplen la **postcondición**.

En el caso de la función `cuadrado` tenemos que:

- El conjunto origen es `float`, ya que su parámetro *n* está declarado de tipo `float` en la signatura de la función.

  Por tanto, los datos de entrada a la función deberán pertenecer al tipo `float`.

- El dominio coincide con el conjunto origen, ya que su precondición es `True`. Eso quiere decir que cualquier dato de entrada es válido siempre que pertenezca al dominio (en este caso, el tipo `float`).
- El conjunto imagen también es `float`, ya que así está declarado el tipo de retorno de la función.

Las pre y postcondiciones no es necesario escribirlas de una manera **formal y rigurosa**, usando el lenguaje de las Matemáticas o la Lógica.

Si la especificación se escribe en *lenguaje natural* y se entiende bien, completamente y sin ambigüedades, no hay problema.

El motivo de usar un lenguaje formal es que, normalmente, resulta **mucho más conciso y preciso que el lenguaje natural**.

El lenguaje natural suele ser:

- **Más prolijo**: necesita más palabras para decir lo mismo que diríamos matemáticamente usando menos caracteres.
- **Más ambiguo**: lo que se dice en lenguaje natural se puede interpretar de distintas formas.
- **Menos completo**: quedan flecos y situaciones especiales que no se tienen en cuenta.

En este otro ejemplo, más completo, se especifica una función llamada `cuenta`:

$$\begin{cases} \textbf{Pre}: & car \neq \texttt{""} \land \texttt{len(}car\texttt{)} = 1 \\ & \texttt{cuenta(}cadena\texttt{: str, }car\texttt{: str) -> int} \\ \textbf{Post}: & \texttt{cuenta(}cadena\texttt{, }car\texttt{)} \geq 0 \land \\ & \texttt{cuenta(}cadena\texttt{, }car\texttt{)} = cadena\texttt{.count(}car\texttt{)} \end{cases}$$

Con esta especificación, estamos diciendo que `cuenta` es una función que recibe una cadena y un carácter (otra cadena con un único carácter dentro).

Ahora bien: esa cadena y ese carácter no pueden ser cualesquiera, sino que tienen que cumplir la *precondición*.

Eso significa, entre otras cosas, que aquí **el *dominio* y el *conjunto origen* de la función no coinciden** (no todos los valores pertenecientes al conjunto origen sirven como datos de entrada válidos para la función).

En esta especificación, `count` se usa como un **método auxiliar**.

Las *operaciones auxiliares* se puede usar en una especificación siempre que estén perfectamente especificadas, aunque no estén implementadas.

En este caso, se usa en la *postcondición* para decir que la función `cuenta`, la que se está especificando, debe devolver el mismo resultado que devuelve el método `count` (el cual ya conocemos perfectamente y sabemos qué hace, puesto que es un método que ya existe en Python).

Es decir: la especificación anterior describe con total precisión que la función `cuenta` **cuenta el número de veces que el carácter *car* aparece en la cadena *cadena***.

En realidad, las condiciones de la especificación anterior se podrían simplificar aprovechando las propiedades de las expresiones lógicas, quedando así:

$$\begin{cases} \textbf{Pre}: & \texttt{len(}car\texttt{)} = 1 \\ & \texttt{cuenta(}cadena\texttt{: str, }car\texttt{: str) -> int} \\ \textbf{Post}: & \texttt{cuenta(}cadena\texttt{, }car\texttt{)} = cadena\texttt{.count(}car\texttt{)} \end{cases}$$

Ejercicio

1. ¿Por qué?

Finalmente, podríamos escribir la misma especificación en lenguaje natural:

$$\begin{cases} \textbf{Pre}: & car \text{ debe ser un único carácter} \\ & \texttt{cuenta(}cadena\texttt{: str, }car\texttt{: str) -> int} \\ \textbf{Post}: & \texttt{cuenta(}cadena\texttt{, }car\texttt{)} \text{ devuelve el número de veces} \\ & \text{que aparece el carácter } car \text{ en la cadena } cadena. \\ & \text{Si } cadena \text{ es vacía o } car \text{ no aparece nunca en la} \\ & \text{cadena } cadena, \text{ debe devolver 0.} \end{cases}$$

Probablemente resulta más fácil de leer (sobre todo para los novatos), pero también es más largo y prolijo.

Es como un contrato escrito por un abogado en lenguaje jurídico.

3. Recursividad

3.1. Funciones y procesos

Los **procesos** son entidades abstractas que habitan los ordenadores.

Conforme van evolucionando, los procesos manipulan otras entidades abstractas llamadas **datos**.

La evolución de un proceso está dirigida por un patrón de reglas llamado **programa**.

Los programadores crean programas para **dirigir** a los procesos.

Es como decir que los programadores son magos que invocan a los espíritus del ordenador (los procesos) con sus conjuros (los programas) escritos en un lenguaje mágico (el lenguaje de programación).

Una **función** describe la *evolución local* de un **proceso**, es decir, cómo se debe comportar el proceso durante la ejecución de la función.

En cada paso de la ejecución se calcula el *siguiente estado* del proceso basándonos en el estado actual y en las reglas definidas por la función.

Nos gustaría ser capaces de visualizar y de realizar afirmaciones sobre el comportamiento global del proceso cuya evolución local está definida por la función.

Esto, en general, es muy difícil, pero al menos vamos a describir algunos de los modelos típicos de evolución de los procesos.

3.1.1. Funciones *ad-hoc*

Supongamos que queremos diseñar una función llamada `permutas` que reciba un número entero *n* y que calcule cuántas permutaciones distintas podemos hacer con *n* elementos.

Por ejemplo: si tenemos 3 elementos (digamos, *A*, *B* y *C*), podemos formar con ellos las siguientes permutaciones:

$$ABC, ACB, BAC, BCA, CAB, CBA$$

y, por tanto, con 3 elementos podemos formar 6 permutaciones distintas. En consecuencia, `permutas(3)` debe devolver `6`.

La implementación de esa función deberá satisfacer la siguiente especificación:

$$\begin{cases} \textbf{Pre}: & n \geq 0 \\ & \texttt{permutas(}n\texttt{: int) -> int} \\ \textbf{Post}: & \texttt{permutas(}n\texttt{)} = \text{el número de permutaciones que} \\ & \text{podemos formar con } n \text{ elementos} \end{cases}$$

Un programador con poca idea de programación (o muy listillo) se podría plantear una implementación parecida a la siguiente:

```
permutas = lambda n: 1 if n == 0 else 1 if n == 1 else 2 if n == 2 else ...
```

que se puede escribir mejor usando la barra invertida (\) para poder separar una instrucción en varias líneas:

```
permutas = lambda n: 1 if n == 0 else \
                     1 if n == 1 else \
                     2 if n == 2 else \
                     6 if n == 3 else \
                     24 if n == 4 else \
                     ...                    # sigue y sigue
```

Pero este algoritmo en realidad es *tramposo*, porque no calcula nada, sino que se limita a asociar el dato de entrada con el de salida, que se ha tenido que calcular previamente usando otro procedimiento.

Este tipo de algoritmos se denominan **algoritmos *ad-hoc***, y las funciones que los implementan se denominan **funciones *ad-hoc***.

Las funciones *ad-hoc* **no son convenientes** porque:

- Realmente son **tramposos** (no calculan nada).
- **No son útiles**, porque al final el cálculo se tiene que hacer con otra cosa.

- Generalmente resulta **imposible** que una función de este tipo abarque todos los posibles datos de entrada, ya que, en principio, puede haber **infinitos** y, por tanto, su código fuente también tendría que ser infinito.

Usar algoritmos y funciones *ad-hoc* se penaliza en esta asignatura.

3.2. Funciones recursivas

3.2.1. Definición

Una **función recursiva** es aquella que se define en términos de sí misma.

Eso quiere decir que, durante la ejecución de una llamada a la función, se ejecuta otra llamada a la misma función, es decir, que la función se llama a sí misma directa o indirectamente.

La forma más sencilla y habitual de función recursiva es aquella en la que **la propia definición de la función contiene una o varias llamadas a ella misma**. En tal caso, decimos que la función se llama a sí misma *directamente* o que hay una **recursividad directa**.

Ese es el tipo de recursividad que vamos a estudiar.

Las definiciones recursivas son el mecanismo básico para ejecutar **repeticiones de instrucciones** en un lenguaje de programación funcional.

Por ejemplo:

$$f(n) = n + f(n+1)$$

Esta función matemática es *recursiva* porque aparece ella misma en su propia definición.

Para calcular el valor de $f(n)$ tenemos que volver a utilizar la propia función f.

Por ejemplo:

$$f(1) = 1 + f(2) = 1 + 2 + f(3) = 1 + 2 + 3 + f(4) = \dots$$

Cada vez que una función se llama a sí misma decimos que se realiza una **llamada recursiva** o **paso recursivo**.

Ejercicio

2. Desde el principio del curso ya hemos estado trabajando con estructuras que pueden tener una definición recursiva. ¿Cuáles son?

3.2.2. Casos base y casos recursivos

Resulta importante que una definición recursiva se detenga alguna vez y proporcione un resultado, ya que si no, no sería útil (tendríamos lo que se llama una **recursión infinita**).

Por tanto, en algún momento, la recursión debe alcanzar un punto en el que la función no se llame a sí misma y se detenga.

Para ello, es necesario que la función, en cada paso recursivo, se vaya acercando cada vez más a ese punto.

Ese punto en el que la función recursiva **no se llama a sí misma**, se denomina **caso base**, y puede haber más de uno.

Los casos base, por tanto, determinan bajo qué condiciones la función no se llamará a sí misma, o dicho de otra forma, con qué valores de sus argumentos la función devolverá directamente un valor y no provocará una nueva llamada recursiva.

Los demás casos, que sí provocan llamadas recursivas, se denominan **casos recursivos**.

3.2.3. El factorial

El ejemplo más típico de función recursiva es el **factorial**.

El factorial de un número natural n se representa por $n!$ y se define como el producto de todos los números desde 1 hasta n:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 1$$

Por ejemplo:

$$6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$$

Pero para calcular 6! también se puede calcular 5! y después multiplicar el resultado por 6, ya que:

$$6! = 6 \cdot \overbrace{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}^{5!}$$

$$6! = 6 \cdot 5!$$

Por tanto, el factorial se puede definir de forma **recursiva**.

Tenemos el **caso recursivo**, pero necesitamos al menos un **caso base** para evitar que la recursión se haga *infinita*.

El caso base del factorial se obtiene sabiendo que el factorial de 0 es directamente 1 (no hay que llamar al factorial recursivamente):

$$0! = 1$$

Combinando ambos casos tendríamos:

$$n! = \begin{cases} 1 & \text{si } n = 0 \quad \text{(caso base)} \\ n \cdot (n-1)! & \text{si } n > 0 \quad \text{(caso recursivo)} \end{cases}$$

La **especificación** de una función que calcule el factorial de un número sería:

$$\begin{cases} \textbf{Pre}: & n \geq 0 \\ & \texttt{factorial(}n\texttt{: int) -> int} \\ \textbf{Post}: & \texttt{factorial(}n\texttt{)} = n! \end{cases}$$

Y su **implementación** en Python podría ser la siguiente:

```python
factorial = lambda n: 1 if n == 0 else n * factorial(n - 1)
```

que sería prácticamente una traducción literal de la definición recursiva de factorial que acabamos de obtener.

3.2.4. Diseño de funciones recursivas

El diseño de funciones recursivas se basa en:

1. Identificación de casos base
2. Descomposición (reducción) del problema
3. Pensamiento optimista

3.2.4.1. Identificación de casos base

Debemos identificar los ejemplares para los cuales hay una solución directa que no necesita recursividad.

Esos ejemplares representarán los *casos base* de la función recursiva, y por eso los denominamos *ejemplares básicos*.

Por ejemplo:

- Supongamos que queremos diseñar una función (llamada *fact*, por ejemplo) que calcule el factorial de un número.

  Es decir: $fact(n)$ debe devolver el factorial de n.
- Sabemos que $0! = 1$, por lo que nuestra función podría devolver directamente 1 cuando se le pida calcular el factorial de 0.
- Por tanto, el caso base del factorial es el cálculo del factorial de 0:

$$fact(0) = 1$$

3.2.4.2. Descomposición (reducción) del problema

Reducimos el problema de forma que así tendremos un ejemplar *más pequeño* del problema.

Un ejemplar más pequeño es aquel que está **más cerca del caso base**.

De esta forma, cada ejemplar se irá acercando más y más al caso base hasta que finalmente se alcanzará dicho caso base y eso detendrá la recursión.

Es importante comprobar que eso se cumple, es decir, que la reducción que le realizamos al problema produce ejemplares que están más cerca del caso base, porque de lo contrario se produciría una *recursión infinita*.

En el ejemplo del factorial:

- El caso base es $fact(0)$, es decir, el caso en el que queremos calcular el factorial de 0, que ya vimos que es directamente 1 (sin necesidad de llamadas recursivas).

- Si queremos resolver el problema de calcular, por ejemplo, el factorial de 5, podríamos intentar reducir el problema a calcular el factorial de 4, que es un número que está más cerca del caso base (que es 0).
- A su vez, para calcular el factorial de 4, reduciríamos el problema a calcular el factorial de 3, y así sucesivamente.
- De esta forma, podemos reducir el problema de calcular el factorial de *n* a calcular el factorial de $(n-1)$, que es un número que está más cerca del 0. Así, cada vez estaremos más cerca del caso base y, al final, siempre lo acabaremos alcanzando.

3.2.4.3. Pensamiento optimista

Consiste en suponer que la función deseada ya existe y que, aunque no sabe resolver el ejemplar original del problema, sí que es capaz de resolver ejemplares *más pequeños* de ese problema (este paso se denomina **hipótesis inductiva** o **hipótesis de inducción**).

Suponiendo que se cumple la *hipótesis inductiva*, y aprovechando que ya contamos con un método para *reducir el ejemplar a uno más pequeño*, ahora tratamos de encontrar un *patrón común* de forma que resolver el ejemplar original implique usar el mismo patrón en un ejemplar más pequeño.

Es decir:

- Al reducir el problema, obtenemos un ejemplar más pequeño del mismo problema y, por tanto, podremos usar la función para poder resolver ese ejemplar más pequeño (que sí sabe resolverlo, por hipótesis inductiva).
- A continuación, usamos dicha solución *parcial* para tratar de obtener la solución para el ejemplar original del problema.

En el ejemplo del factorial:

- Supongamos que queremos calcular, por ejemplo, el factorial de 6.
- Aún no sabemos calcular el factorial de 6, pero suponemos (por *hipótesis inductiva*) que sí sabemos calcular el factorial de 5.

  En ese caso, ¿cómo puedo aprovechar que sé resolver el factorial de 5 para lograr calcular el factorial de 6?
- Analizando el problema, observo que se cumple esta propiedad:

$$6! = 6 \cdot \overbrace{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}^{5!} = 6 \cdot 5!$$

  Por tanto, he deducido un método para resolver el problema de calcular el factorial de 6 a partir del factorial de 5: *para calcular el factorial de 6 basta con calcular primero el factorial de 5 y luego multiplicar el resultado por 6.*

> Dicho de otro modo: *si yo supiera* calcular el factorial de 5, me bastaría con multiplicarlo por 6 para obtener el factorial de 6.

Generalizando para cualquier número, no sólo para el 6:

- Si queremos diseñar una función $fact(n)$ que calcule el factorial de n, supondremos que esa función ya existe pero que aún no sabe calcular el factorial de n, aunque **sí sabe calcular el factorial de $(n-1)$**.

  Tenemos que creer en que es así y actuar como si fuera así, aunque ahora mismo no sea verdad. *Ésta es nuestra **hipótesis inductiva**.*

- Por otra parte, sabemos que:

$$n! = n \cdot \overbrace{(n-1) \cdot (n-2) \cdot (n-3) \cdot 2 \cdot 1}^{(n-1)!} = n \cdot (n-1)!$$

  Por tanto, si sabemos calcular el factorial de $(n-1)$ llamando a $fact(n-1)$, para calcular $fact(n)$ sólo necesito multiplicar n por el resultado de $fact(n-1)$.

  > Resumiendo: ***si yo supiera* calcular el factorial de $(n-1)$, me bastaría con multiplicarlo por n para obtener el factorial de n.**

- Así obtengo el caso recursivo de la función *fact*, que sería:

$$fact(n) = n \cdot fact(n-1)$$

Combinando todos los pasos, obtenemos la solución general:

$$fact(n) = \begin{cases} 1 & \text{si } n = 0 \quad \text{(caso base)} \\ n \cdot fact(n-1) & \text{si } n > 0 \quad \text{(caso recursivo)} \end{cases}$$

3.2.5. Recursividad lineal

Una función tiene **recursividad lineal** si cada llamada a la función recursiva genera, como mucho, otra llamada recursiva a la misma función.

El factorial definido en el ejemplo anterior es un caso típico de recursividad lineal ya que, cada vez que se llama al factorial se genera, como mucho, otra llamada al factorial.

Eso se aprecia claramente observando que la definición del caso recursivo de la función *fact* contiene una única llamada a la misma función *fact*:

$$fact(n) = n \cdot fact(n-1) \quad \text{si } n > 0 \quad \text{(caso recursivo)}$$

3.2.5.1. Procesos recursivos lineales

La forma más directa y sencilla de definir una función que calcule el factorial de un número a partir de su definición recursiva podría ser la siguiente:

```
factorial = lambda n: 1 if n == 0 else n * factorial(n - 1)
```

Utilizaremos el modelo de sustitución para observar el funcionamiento de esta función al calcular 6!:

```
factorial(6)
= (6 * factorial(5))
= (6 * (5 * factorial(4)))
= (6 * (5 * (4 * factorial(3))))
= (6 * (5 * (4 * (3 * factorial(2)))))
= (6 * (5 * (4 * (3 * (2 * factorial(1))))))
= (6 * (5 * (4 * (3 * (2 * (1 * factorial(0)))))))
= (6 * (5 * (4 * (3 * (2 * (1 * 1))))))
= (6 * (5 * (4 * (3 * (2 * 1)))))
= (6 * (5 * (4 * (3 * 2))))
= (6 * (5 * (4 * 6)))
= (6 * (5 * 24))
= (6 * 120)
= 720
```

Podemos observar un perfil de **expansión** seguido de una **contracción**:

- La **expansión** ocurre conforme el proceso construye una secuencia de operaciones a realizar *posteriormente* (en este caso, una secuencia de multiplicaciones).
- La **contracción** se realiza conforme se van ejecutando realmente las multiplicaciones.

Llamaremos **proceso recursivo** a este tipo de proceso caracterizado por una secuencia de **operaciones pendientes de completar**.

Para poder ejecutar este proceso, el intérprete necesita **memorizar**, en algún lugar, un registro de las multiplicaciones que se han dejado para más adelante.

En el cálculo de $n!$, la longitud de la secuencia de operaciones pendientes (y, por tanto, la información que necesita almacenar el intérprete), crece *linealmente* con n, al igual que el número de pasos de reducción.

A este tipo de procesos lo llamaremos **proceso recursivo *lineal***.

3.2.5.2. Procesos iterativos lineales

A continuación adoptaremos un enfoque diferente.

Podemos mantener un producto acumulado y un contador desde n hasta 1, de forma que el contador y el producto cambien de un paso al siguiente según la siguiente regla:

$$acumulador_{nuevo} = acumulador_{viejo} \cdot contador_{viejo}$$
$$contador_{nuevo} = contador_{viejo} - 1$$

Su traducción a Python podría ser la siguiente, usando una función auxiliar `fact_iter`:

```
fact_iter = lambda cont, acc: acc if cont == 0 else \
                              fact_iter(cont - 1, cont * acc)
factorial = lambda n: fact_iter(n, 1)
```

Al igual que antes, usaremos el modelo de sustitución para visualizar el proceso del cálculo de 6!:

```
factorial(6)
= fact_iter(6, 1)
= fact_iter(5, 6)
= fact_iter(4, 30)
= fact_iter(3, 120)
= fact_iter(2, 360)
= fact_iter(1, 720)
= fact_iter(0, 720)
= 720
```

Este proceso no tiene expansiones ni contracciones ya que, en cada instante, toda la información que se necesita almacenar es el valor actual de los parámetros `cont` y `acc`, por lo que el tamaño de la memoria necesaria es constante.

A este tipo de procesos lo llamaremos **proceso iterativo**.

El número de pasos necesarios para calcular *n*! usando esta función crece *linealmente* con *n*.

A este tipo de procesos lo llamaremos **proceso iterativo *lineal***.

Tipo de proceso	Número de reducciones	Memoria necesaria
Recursivo	Proporcional a *n*	Proporcional a *n*
Iterativo	Proporcional a *n*	Constante

Tipo de proceso	Número de reducciones	Memoria necesaria
Recursivo lineal	Linealmente proporcional a *n*	Linealmente proporcional a *n*
Iterativo lineal	Linealmente proporcional a *n*	Constante

En general, un **proceso iterativo** es aquel que está definido por una serie de **coordenadas de estado** junto con una **regla** fija que describe cómo actualizar dichas coordenadas conforme cambia el proceso de un estado al siguiente.

La **diferencia entre los procesos recursivo e iterativo** se puede describir de esta otra manera:

- En el **proceso iterativo**, los parámetros dan una descripción completa del estado del proceso en cada instante.

  Así, si parásemos el cálculo entre dos pasos, lo único que necesitaríamos hacer para seguir con el cálculo es darle al intérprete el valor de los dos parámetros.

- En el **proceso recursivo**, el intérprete tiene que mantener cierta información *oculta* que no está almacenada en ningún parámetro y que indica qué operaciones ha realizado hasta ahora y cuáles quedan pendientes por hacer.

No debe confundirse un **proceso recursivo** con una **función recursiva**:

- Cuando hablamos de *función recursiva* nos referimos al hecho de que la función se llama a sí misma (directa o indirectamente).
- Cuando hablamos de *proceso recursivo* nos referimos a la forma en como se desenvuelve la ejecución de la función (con una expansión más una contracción).

Puede parecer extraño que digamos que una función recursiva (por ejemplo, `fact_iter`) genera un proceso iterativo.

Sin embargo, el proceso es realmente iterativo porque su estado está definido completamente por dos parámetros, y para ejecutar el proceso sólo se necesita almacenar el valor de esos dos parámetros.

Aquí hemos visto un ejemplo donde se aprecia claramente que **una función sólo puede tener una especificación** pero **puede tener varias implementaciones** distintas.

Eso sí: todas las implementaciones de una función deben satisfacer su especificación.

En este caso, las dos implementaciones son:

```
factorial = lambda n: 1 if n == 0 else n * factorial(n - 1)
```

y

```
fact_iter = lambda cont, acc: acc if cont == 0 else \
                              fact_iter(cont - 1, cont * acc)
factorial = lambda n: fact_iter(n, 1)
```

Y aunque las dos satisfacen la misma especificación (y, por tanto, calculan exactamente los mismos valores), lo hacen de una forma muy diferente, generando incluso procesos de distinto tipo.

3.2.6. Recursividad múltiple

Una función tiene **recursividad múltiple** cuando, durante la misma activación o llamada a la función, se puede generar más de una llamada recursiva a la misma función.

El ejemplo clásico es la función que calcula los términos de la **sucesión de Fibonacci**.

La sucesión comienza con los números 0 y 1, y a partir de éstos, cada término es la suma de los dos anteriores:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ...

Podemos definir una función recursiva que devuelva el n-ésimo término de la sucesión de Fibonacci:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \quad \text{(caso base)} \\ 1 & \text{si } n = 1 \quad \text{(caso base)} \\ fib(n-1) + fib(n-2) & \text{si } n > 1 \quad \text{(caso recursivo)} \end{cases}$$

La especificación de una función que devuelva el n-ésimo término de la sucesión de Fibonacci sería:

$$\begin{cases} \textbf{Pre}: & n \geq 0 \\ & \texttt{fib(}n\texttt{: int) -> int} \\ \textbf{Post}: & \texttt{fib(}n\texttt{)} = \text{el } n\text{-ésimo término de la sucesión de Fibonacci} \end{cases}$$

Y su implementación en Python podría ser:

```
fib = lambda n: 0 if n == 0 else 1 if n == 1 else fib(n - 1) + fib(n - 2)
```

o bien, separando la definición en varias líneas:

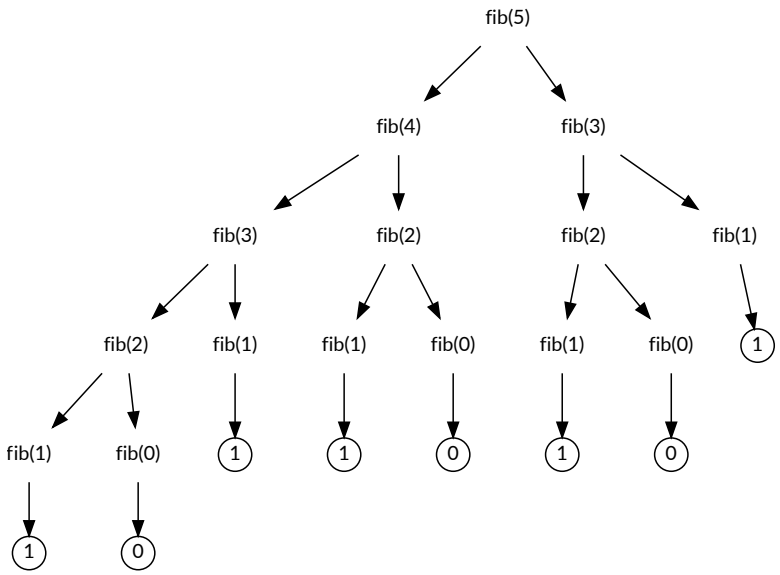
```
fib = lambda n: 0 if n == 0 else \
                1 if n == 1 else \
                fib(n - 1) + fib(n - 2)
```

Si vemos el perfil de ejecución de `fib(5)`, vemos que:

- Para calcular `fib(5)`, antes debemos calcular `fib(4)` y `fib(3)`.
- Para calcular `fib(4)`, antes debemos calcular `fib(3)` y `fib(2)`.
- Así sucesivamente hasta poner todo en función de `fib(0)` y `fib(1)`, que se pueden calcular directamente (son los casos base).

En general, el proceso resultante tiene forma de árbol.

Por eso decimos que las funciones con recursividad múltiple generan **procesos recursivos en árbol**.



La función anterior es un buen ejemplo de recursión en árbol, pero desde luego es un método *horrible* para calcular los números de Fibonacci, por la cantidad de **operaciones redundantes** que efectúa.

Para tener una idea de lo malo que es, se puede observar que $fib(n)$ crece exponencialmente en función de n.

Por lo tanto, el proceso necesita una cantidad de tiempo que crece **exponencialmente** con n.

Por otro lado, el espacio necesario sólo crece **linealmente** con n, porque en un cierto momento del cálculo sólo hay que memorizar los nodos que hay por encima.

En general, en un proceso recursivo en árbol **el tiempo de ejecución crece con el *número de nodos* del árbol** mientras que **el espacio necesario crece con la *altura máxima* del árbol**.

Se puede construir un **proceso iterativo** para calcular los números de Fibonacci.

La idea consiste en usar dos coordenadas de estado a y b (con valores iniciales 0 y 1, respectivamente) y aplicar repetidamente la siguiente transformación:

$$a_{nuevo} = b_{viejo}$$
$$b_{nuevo} = b_{viejo} + a_{viejo}$$

Después de n pasos, a y b contendrán $fib(n)$ y $fib(n+1)$, respectivamente.

En Python sería:

```
fib_iter = lambda cont, a, b: a if cont == 0 else fib_iter(cont - 1, b, a + b)
fib = lambda n: fib_iter(n, 0, 1)
```

Esta función genera un proceso iterativo lineal, por lo que es mucho más eficiente.

3.2.7. Recursividad final y no final

Lo que diferencia al `fact_iter` que genera un proceso iterativo del `factorial` que genera un proceso recursivo, es el hecho de que `fact_iter` se llama a sí misma y devuelve directamente el valor que le ha devuelto su llamada recursiva sin hacer luego nada más.

En cambio, `factorial` tiene que hacer una multiplicación después de llamarse a sí misma y antes de terminar de ejecutarse:

```
# Versión con recursividad final:
fact_iter = lambda cont, acc: acc if cont == 0 else \
                              fact_iter(cont - 1, acc * cont)
fact = lambda n: fact_iter(n, 1)

# Versión con recursividad no final:
factorial = lambda n: 1 if n == 0 else n * factorial(n - 1)
```

Es decir:

- `fact_iter(cont, acc)` simplemente llama a:

  `fact_iter(cont - 1, acc * cont)`

  y luego devuelve directamente el valor que le entrega ésta llamada, sin hacer ninguna otra operación posterior antes de terminar.

- En cambio, `factorial(n)` hace:

  ```
  n * factorial(n - 1)
  ```

  o sea, se llama a sí misma pero el resultado de la llamada recursiva tiene que multiplicarlo luego por `n` antes de devolver el resultado final.

Por tanto, **lo último que hace `fact_iter` es llamarse a sí misma**. En cambio, lo último que hace `factorial` no es llamarse a sí misma, porque tiene que hacer más operaciones (en este caso, la multiplicación) antes de devolver el resultado.

Cuando lo último que hace una función recursiva es llamarse a sí misma y devolver directamente el valor devuelto por esa llamada recursiva, decimos que la función es **recursiva final** o que tiene **recursividad final**.

En caso contrario, decimos que la función es **recursiva no final** o que tiene **recursividad no final**.

Las funciones recursivas finales generan procesos iterativos.

La función `fact_iter` es recursiva final, y por eso genera un proceso iterativo.

En cambio, la función `factorial` es recursiva no final, y por eso genera un proceso recursivo.

En la práctica, para que un proceso iterativo consuma realmente una cantidad constante de memoria, es necesario que el traductor **optimice la recursividad final**.

Ese tipo de optimización se denomina ***tail-call optimization (TCO)***.

No muchos traductores optimizan la recursividad final.

De hecho, ni el intérprete de Python ni la máquina virtual de Java optimizan la recursividad final.

Por tanto, en estos dos lenguajes, las funciones recursivas finales consumen tanta memoria como las no finales.

3.3. Tipos de datos recursivos

3.3.1. Concepto

Un **tipo de dato recursivo** es aquel que puede definirse en términos de sí mismo.

Un **dato recursivo** es un dato que pertenece a un tipo recursivo. Por tanto, es un dato que se construye sobre otros datos del mismo tipo.

Como toda estructura recursiva, un tipo de dato recursivo tiene casos base y casos recursivos:

- En los casos base, el tipo recursivo se define directamente, sin referirse a sí mismo.
- En los casos recursivos, el tipo recursivo se define sobre sí mismo.

La forma más natural de manipular un dato recursivo es usando funciones recursivas.

3.3.2. Cadenas

Las **cadenas** se pueden considerar **tipos de datos recursivos**, ya que podemos decir que toda cadena `c`:

- o bien es la cadena vacía `''` (*caso base*),
- o bien está formada por dos partes:
  - El **primer carácter** de la cadena, al que se accede mediante `c[0]`.
  - El **resto** de la cadena (al que se accede mediante `c[1:]`), que también es una cadena (*caso recursivo*).

  En tal caso, se cumple que `c == c[0] + c[1:]`.

Eso significa que podemos acceder al segundo carácter de la cadena (suponiendo que exista) mediante `c[1:][0]`.

```
cadena = 'hola'
cadena[0]       # devuelve 'h'
cadena[1:]      # devuelve 'ola'
cadena[1:][0]   # devuelve 'o'
```

3.3.3. Tuplas

Las **tuplas** (datos de tipo `tuple`) son una generalización de las cadenas.

Una tupla es una **secuencia de elementos** que no tienen por qué ser caracteres, sino que cada uno de ellos pueden ser **de cualquier tipo** (números, cadenas, booleanos, ..., incluso otras tuplas).

Los literales de tipo tupla se representan enumerando sus elementos separados por comas y encerrados entre paréntesis.

Por ejemplo:

```
tupla = (27, 'hola', True, 73.4, ('a', 'b', 'c'), 99)
```

Si sólo tiene un elemento, hay que poner una coma detrás:

```
tupla = (35,)
```

Las tuplas también pueden verse como un **tipo de datos recursivo**, ya que toda tupla `t`:

- o bien es la tupla vacía, representada mediante `()` (*caso base*),
- o bien está formada por dos partes:
  - El **primer elemento** de la tupla (al que se accede mediante `t[0]`), que hemos visto que puede ser de cualquier tipo.
  - El **resto** de la tupla (al que se accede mediante `t[1:]`), que también es una tupla (*caso recursivo*).

Según el ejemplo anterior:

```
>>> tupla = (27, 'hola', True, 73.4, ('a', 'b', 'c'), 99)
>>> tupla[0]
27
>>> tupla[1:]
('hola', True, 73.4, ('a', 'b', 'c'), 99)
>>> tupla[1:][0]
'hola'
```

Junto a las operaciones `t[0]` y `t[1:]`, tenemos también la operación `+` (**concatenación**), al igual que ocurre con las cadenas.

Con la concatenación se pueden crear nuevas tuplas a partir de otras tuplas.

Por ejemplo:

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
```

Eso significa que, si `t` es una tupla no vacía, se cumple que `t == (t[0],) + t[1:]`.

Esta propiedad es similar (aunque no exactamente igual) a la que se cumple en las cadenas no vacías.

3.3.4. Rangos

Los rangos (datos de tipo `range`) son valores que representan **sencuencias de números enteros**.

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.

Ejemplos

`range(10)` representa la secuencia $0, 1, 2, \ldots, 9$.

`range(3, 10)` representa la secuencia $3, 4, 5, \ldots, 9$.

`range(0, 10, 2)` representa la secuencia $0, 2, 4, 6, 8$.

`range(4, 0, -1)` representa la secuencia $4, 3, 2, 1$.

`range(3, 3)` representa el rango vacío.

`range(4, 3)` también representa el rango vacío.

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(2, 3)
range(2, 3)
>>> range(4)
range(0, 4)
```

```
>>> range(2, 5, 1)
range(2, 5)
>>> range(2, 5, 2)
range(2, 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 también pueden verse como un **tipo de datos recursivo**, ya que todo rango `r`:

- o bien es el rango vacío (*caso base*),
- o bien está formado por dos partes:
  - El **primer elemento** del rango (al que se accede mediante `r[0]`), que hemos visto que tiene que ser un número entero.
  - El **resto** del rango (al que se accede mediante `r[1:]`), que también es un rango (*caso recursivo*).

Según el ejemplo anterior:

```
>>> rango = range(4, 7)
>>> rango[0]
4
>>> rango[1:]
range(5, 7)
>>> rango[1:][0]
5
```

3.3.5. Conversión a tupla

Las cadenas y los rangos se pueden convertir fácilmente a tuplas usando la función `tuple`:

```
>>> tuple('hola')
('h', 'o', 'l', 'a')
>>> tuple('')
()
```

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

Bibliografía

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, Mass. : New York: MIT Press ; McGraw-Hill.