

# Abstracciones funcionales

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/08/05 a las 11:14:00

## Índice

<b>1. Abstracciones lambda</b>	<b>2</b>
1.1. Expresiones lambda . . . . .	2
1.2. Parámetros y cuerpos . . . . .	2
1.3. Aplicación funcional . . . . .	3
1.3.1. Evaluación de una aplicación funcional . . . . .	3
1.3.2. Funciones con nombre . . . . .	4
1.4. Identificadores cuantificados y libres de una expresión lambda . . . . .	5
<b>2. Ámbitos</b>	<b>6</b>
2.1. Ámbitos léxicos . . . . .	6
2.2. Ámbito de una definición y de una ligadura . . . . .	9
2.2.1. Almacenamiento . . . . .	10
2.2.2. Visibilidad . . . . .	11
2.2.3. Tiempo de vida . . . . .	11
2.3. Ámbito de un identificador . . . . .	14
2.4. Ámbito de un parámetro . . . . .	15
2.5. Ámbito de un identificador cuantificado . . . . .	18
2.6. Ámbito de un identificador libre . . . . .	19
<b>3. Funciones recursivas</b>	<b>19</b>
3.1. Definición . . . . .	19
3.2. Casos base y casos recursivos . . . . .	20
3.3. El factorial . . . . .	20
3.4. Diseño de funciones recursivas . . . . .	21
3.4.1. Identificación de casos base . . . . .	21
3.4.2. Descomposición (reducción) del problema . . . . .	22
3.4.3. Pensamiento optimista . . . . .	22
3.5. Recursividad lineal . . . . .	24
3.5.1. Procesos recursivos lineales . . . . .	24
3.5.2. Procesos iterativos lineales . . . . .	25
3.6. Recursividad múltiple . . . . .	27
3.7. Recursividad final y no final . . . . .	29

4. Abstracciones funcionales	30
4.1. Las funciones como abstracciones y generalizaciones . . . . .	30
4.2. Especificaciones de funciones . . . . .	34

## 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  **$\beta$ -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  $\beta$ -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 ( $\beta$ -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

```

## 1.4. 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
```

The diagram illustrates the binding of variables in the lambda expression. A bracket under the variable `x` is labeled "parámetro", and a bracket under the variable `y` is labeled "identificador cuantificado".

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. Ámbitos

### 2.1. Ámbitos léxicos

Un **ámbito léxico** (también llamado **ámbito estático**) es una porción del código fuente de un programa.

Decimos que **ciertas construcciones sintácticas determinan ámbitos léxicos**.

Cuando una construcción determina un ámbito léxico, la **sintaxis del lenguaje establece dónde empieza y acaba** ese ámbito léxico en el código fuente.

Por tanto, siempre se puede determinar sin ambigüedad si **una instrucción situada en un punto concreto del programa está dentro de un determinado ámbito léxico**, tan sólo leyendo el código fuente del programa y sin necesidad de ejecutarlo.

Eso significa que el concepto de *ámbito léxico* es un concepto **estático**.

Por ejemplo: en el lenguaje de programación Java, los *bloques* son estructuras sintácticas delimitadas por llaves { y } que contienen instrucciones.

Los bloques de Java determinan ámbitos léxicos; por tanto, si una instrucción está dentro de un bloque (es decir, si está situada entre las llaves { y } que delimitan el bloque), entonces esa instrucción se encuentra dentro del ámbito léxico que define el bloque.

Además de los ámbitos léxicos, existen también los llamados **ámbitos dinámicos**, que funcionan de otra forma y que no estudiaremos en este curso.

La mayoría de los lenguajes de programación usan ámbitos léxicos, salvo excepciones (como LISP o los *shell scripts*) que usan ámbitos dinámicos.

Por esa razón, a partir de ahora, cuando hablemos de «ámbitos» sin especificar de qué tipo, nos estaremos siempre refiriendo a «ámbitos léxicos».

Los ámbitos **se pueden anidar recursivamente**, o sea, que pueden estar contenidos unos dentro de otros.

Por tanto, una instrucción puede estar en varios ámbitos al mismo tiempo (anidados unos dentro de otros).

De todos ellos, el **ámbito más interno** es el que no contiene, a su vez, a ningún otro ámbito.

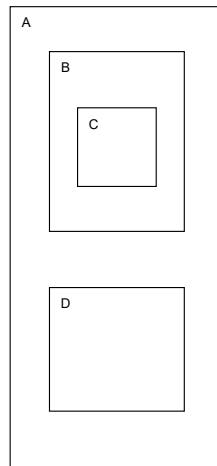
Definimos el **ámbito de una instrucción** como el ámbito más interno en el que se encuentra dicha instrucción.

Según lo anterior, en un momento dado, el **ámbito actual** es el ámbito de la instrucción actual, es decir, el ámbito más interno en el que se encuentra la instrucción que se está ejecutando actualmente.

Decimos que los ámbitos léxicos cumplen la **propiedad de la estructura**.

Una **estructura** es una construcción sintáctica que puede **anidarse completamente** dentro de otras estructuras, de forma que, dadas dos estructuras cualesquiera, o una está incluida completamente dentro de la otra, o no se tocan en absoluto.

Por tanto, los bordes de dos ámbitos léxicos nunca pueden cruzarse:



Ámbitos léxicos anidados

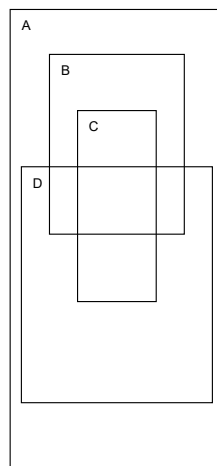
A puede ver los nombres definidos en A, pero no los definidos en B, C o D.

B puede ver los nombres definidos en A y B, pero no los definidos en C o D.

C puede ver los nombres definidos en A, B y C, pero no los definidos en D.

D puede ver los nombres definidos en A y D, pero no los definidos en B o C.

En cambio, los bordes de los siguientes «ámbitos» se cruzan y, por consiguiente, no poseen la propiedad de la estructura:



Estructuras que se cruzan



Por esta razón, estas estructuras no podrían ser nunca ámbitos léxicos.

El concepto de *ámbito* no es nada trivial y, a medida que vayamos incorporando nuevos elementos al lenguaje, tendremos que ir adaptándolo para tener en cuenta más condicionantes.

Hasta ahora sólo hemos tenido un ámbito, llamado **ámbito global**:

- Si se está ejecutando un *script* en el intérprete por lotes (con `python script.py`), el *ámbito global* abarca todo el *script*, desde la primera instrucción hasta la última.
- Si estamos en el intérprete interactivo (con `python` o `ipython3`), el *ámbito global* abarca toda nuestra sesión con el intérprete, desde que arrancamos la sesión hasta que finalizamos la misma.

Por tanto:

- En el momento en que se empieza a ejecutar un *script* o se arranca una sesión con el intérprete interactivo, **se entra** en el *ámbito global*.
- Del ámbito global sólo **se sale** cuando se finaliza la ejecución del *script* o se cierra el intérprete interactivo.

## 2.2. Ámbito de una definición y de una ligadura

El **ámbito de una definición** es el ámbito actual de esa definición, es decir, el ámbito más interno donde aparece esa definición.

Por extensión, llamamos **ámbito de una ligadura** al ámbito de la definición que, al ejecutarse, creará la ligadura (es decir, el ámbito más interno donde aparece la definición que, al ejecutarse, creará la ligadura en tiempo de ejecución).

En la práctica, es lo mismo hablar del «ámbito de una definición» que del «ámbito de la ligadura que se creará al ejecutar la definición», ya que son la misma cosa.

Decimos que la *definición* (y la *ligadura* correspondiente que se creará al ejecutar esa definición) es **local** a su ámbito.

Si ese ámbito es el ámbito *global*, decimos que la *definición* (y la *ligadura* que se creará al ejecutar esa definición) es **global**.

Por ejemplo, en el siguiente *script* se ejecutan cuatro definiciones:

```
x = 25
y = 99
z = y
nombre = "Manolo"
```

Ámbito global

El ámbito de cada una de las instrucciones es el ámbito *global*, que es el único ámbito que existe en el *script*.

En consecuencia:

- Las cuatro definiciones tienen **ámbito global** (y son, por tanto, **definiciones globales**).
- Cuando se ejecuten, esas definiciones crearán **ligaduras globales**.

Como estamos usando un lenguaje de programación que trabaja con *ámbitos léxicos*, **el ámbito de una definición siempre vendrá determinado por una construcción sintáctica** del lenguaje.

Por tanto:

- Sus *límites* vienen marcados únicamente por la *sintaxis* de la construcción que determina el ámbito de esa definición.
- El ámbito de la definición se puede determinar simplemente leyendo el código fuente del programa, observando dónde empieza y dónde acaba esa construcción, sin tener que ejecutarlo.

Es decir, que se puede determinar de forma *estática*.

### 2.2.1. Almacenamiento

Sabemos que las ligaduras se almacenan en *espacios de nombres*.

En Python, hay dos lugares donde se pueden almacenar ligaduras y, por tanto, **hay dos posibles espacios de nombres: los objetos y los marcos**.

Así que tenemos dos posibilidades:

1. Si el identificador que se está ligando es un *atributo* de un objeto, entonces la ligadura se almacenará en el objeto.
2. En caso contrario, la ligadura se almacenará en un marco, el cual depende del *ámbito actual*.

Veamos cada caso con más detalle.

1. Cuando se crea una ligadura dentro de un objeto en Python usando el operador punto (`.`), **el espacio de nombres será el propio objeto**, ya que los objetos son espacios de nombres en Python. En tal caso, la ligadura asocia un valor con un *atributo* del objeto.

Por ejemplo, si en Python hacemos:

```
import math
math.x = 75
```

estamos creando la ligadura `x`  $\rightarrow$  `75` en el espacio de nombres que representa el módulo `math`, el cual es un objeto en Python y, por tanto, es quien almacena la ligadura.

Así que el espacio de nombres ha sido seleccionado a través del operador punto (`.`) para resolver el atributo dentro del objeto, y no depende del ámbito donde se encuentre la sentencia `math.x = 75`.

Diremos que la ligadura es **local** al objeto.

2. Si la ligadura no se crea dentro de un objeto usando el operador punto (`.`), entonces el espacio de nombres irá asociado al ámbito y, en este caso, **ese espacio de nombres siempre será un marco**.

Ese marco será el que corresponda al *ámbito actual*, es decir, el ámbito más interno en el que se encuentra la instrucción que crea la ligadura.

Cuando el ámbito es el *ámbito global* (y, por tanto, la ligadura se almacena en el marco global), se dice que la ligadura es **global**.

En caso contrario, decimos que es **local** al ámbito, y se almacenará en el marco correspondiente a ese ámbito.

### 2.2.2. Visibilidad

La visibilidad de una ligadura indica en qué lugares del programa es visible y accesible esa ligadura.

Es decir: la visibilidad determina dónde en el programa puede usarse un determinado identificador para acceder al valor al que está ligado.

Al igual que antes, tenemos dos posibilidades, dependiendo de si estamos ligando un atributo de un objeto, o no.

1. Si el identificador ligado es un **atributo de un objeto**, la ligadura sólo será visible dentro del objeto.

En tal caso, decimos que la visibilidad de la ligadura (y del correspondiente atributo ligado) es **local al objeto** que contiene el atributo.

Eso significa que debemos indicar (usando el operador punto (.)) el objeto que contiene a la ligadura para poder acceder a ella, lo que significa que también debemos tener acceso al propio objeto que la contiene.

2. Si el identificador ligado **NO es un atributo de un objeto**, la ligadura sólo será visible dentro del ámbito donde se definió la ligadura.

Ese ámbito representa una «región» cuyas fronteras limitan la porción del código fuente en la que es visible esa ligadura.

En tal caso, decimos que la **visibilidad** de la ligadura es **local a su ámbito**.

Eso significa que **no es posible acceder a esa ligadura fuera de su ámbito**; sólo es visible dentro de él.

En cambio, si el ámbito de la ligadura contiene dentro otro ámbito anidado, sí que podremos acceder a la ligadura dentro de ese ámbito más interno, ya que técnicamente seguiría estando dentro de su ámbito.

Si el ámbito es el global, decimos que la ligadura tiene **visibilidad global**.

### 2.2.3. Tiempo de vida

El **tiempo de vida** de una ligadura representa el periodo de tiempo durante el cual *existe* esa ligadura, es decir, el periodo comprendido desde su creación y almacenamiento en la memoria hasta su posterior destrucción.

En la mayoría de los lenguajes (incluyendo Python y Java), una ligadura **empieza a existir** justo cuando se crea, es decir, en el punto donde se ejecuta la instrucción que define la ligadura.

Por tanto, no es posible *acceder* a esa ligadura *antes* de ese punto, ya que no existe hasta entonces.

Por otra parte, el momento en que una ligadura **deja de existir** depende su almacenamiento:

- Si se almacena en un objeto, es porque la ligadura está ligando un atributo de ese objeto a un determinado valor. En tal caso, la ligadura dejará de existir cuando se elimine el objeto de la memoria, o bien, cuando se elimine el atributo ligado.
- En caso contrario, la ligadura estará almacenada en un marco y, por tanto, dejará de existir allí donde termine el ámbito de la ligadura.

Es importante hacer notar que, en un momento dado, una ligadura puede existir pero no ser visible.

Por ejemplo, si una ligadura local y una global vinculan el mismo identificador, la local «hace sombra» a la global, cosa que estudiaremos con más profundidad posteriormente.

### Ejemplo

En el siguiente ejemplo vemos cómo hay varias definiciones que, al ejecutarse, crearán ligaduras en un determinado ámbito, pero no en un objeto (ya que no se están creando atributos dentro de ningún objeto):

```
1 x = 25
2 y = 99
3 z = y
4 nombre = 'Manolo'
```

Todas esas definiciones son globales y, por tanto, las ligaduras que crean al ejecutarse son ligaduras globales o de ámbito global, y se almacenan en el marco global.

Al no tratarse de atributos de objetos, la visibilidad vendrá determinada por sus ámbitos.

En consecuencia, la visibilidad de todas esas ligaduras será el ámbito global, ya que son ligaduras globales. Por tanto, decimos que su **visibilidad es global**.

Por otra parte, como esas ligaduras no se crean sobre atributos de objetos, empezarán a existir justo donde se crean, y terminarán de existir al final de su ámbito.

Por ejemplo, la ligadura `y → 99` empezará a existir en la línea 2 y terminará al final del *script*, que es donde termina su ámbito (que, en este ejemplo, es el ámbito global).

En consecuencia, el **tiempo de vida** de la ligadura será el periodo comprendido desde su creación (en la línea 2) hasta el final de su ámbito.

Cuando la ligadura se crea sobre un **atributo** de un *objeto* de Python, entonces ese objeto almacenará la ligadura y será, por tanto, su espacio de nombres.

Recordemos que, por ejemplo, cuando importamos un módulo usando la sentencia **import**, podemos acceder al objeto que representa ese módulo usando su nombre, lo que nos permite acceder a sus atributos y crear otros nuevos.

Esos atributos y sus ligaduras correspondientes sólo son visibles cuando accedemos a ellos usando el operador punto (`.`) a través del objeto que lo contiene.

Por tanto, los atributos no son visibles fuera del objeto, y debemos usar el operador punto (`.`) para acceder a ellos (su visibilidad es local al objeto que los contiene).

Por ejemplo:

```
>>> import math
>>> math.pi
3.141592653589793      # El nombre 'pi' es visible dentro del objeto
>>> pi                  # El nombre 'pi' no es visible fuera del objeto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
```

Igualmente, si creamos un nuevo atributo dentro del objeto, la ligadura entre el atributo y su valor sólo existirá en el propio objeto y, por tanto, sólo será visible cuando accedamos al atributo a través del objeto donde se ha creado.

```
>>> import math
>>> math.x = 95          # Creamos un nuevo atributo en el objeto
>>> math.x               # El nombre 'x' es visible dentro del objeto
95
>>> x                    # El nombre 'x' no es visible fuera del objeto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Resumiendo:

- Para poder acceder a un atributo de un objeto, debemos acceder primero al objeto y usar el operador punto (.).
- Por tanto, la **visibilidad** de su ligadura correspondiente no vendrá determinada por un ámbito, sino por el objeto que contiene al atributo (y que, por consiguiente, también contiene a su ligadura).

En tal caso, diremos que la visibilidad es local al objeto que contiene el atributo.

- Por otra parte, el **tiempo de vida** de la ligadura será el tiempo que permanezca el atributo en el objeto, ligado a algún valor.

### 2.2.3.1. Resumen

#### Ámbito (léxico):

Porción del código fuente de un programa. Los límites de ese ámbito sólo vienen determinados por la sintaxis del lenguaje, ya que ciertas construcciones sintácticas determinan su propio ámbito.

#### Ámbito de una definición:

El ámbito actual de la definición; es decir: el ámbito más interno donde aparece la definición.

#### Ámbito de una ligadura:

El ámbito de la instrucción que creará la ligadura en tiempo de ejecución. Por ejemplo, si la instrucción es una definición, se corresponde con el ámbito de la definición.

**Visibilidad de una ligadura:**

Determina dónde es visible una ligadura dentro del programa.

Esa visibilidad depende de si el identificador ligado es un atributo de un objeto o no:

- Si es un atributo de un objeto, la visibilidad lo determina el objeto que contiene la ligadura.
- En caso contrario, la visibilidad lo determina el ámbito de la ligadura.

**Tiempo de vida de una ligadura:**

El periodo de tiempo durante el cual *existe* esa ligadura, es decir, el periodo comprendido desde su creación y almacenamiento en la memoria hasta su posterior destrucción.

Su tiempo de vida empieza siempre en el momento en que se crea la ligadura, y su final depende de si el identificador ligado es un atributo de un objeto o no:

- Si es un atributo de un objeto, el tiempo de vida acabará cuando se destruya el objeto que lo contiene (o cuando se elimine el atributo ligado).
- En caso contrario, el tiempo de vida acabará al final del ámbito de la ligadura.

**Almacenamiento de una ligadura:**

Determina el **espacio de nombres** donde se almacenará la ligadura:

- Si el identificador ligado es un atributo de un objeto, el espacio de nombres será el objeto que lo contiene.
- En caso contrario, el espacio de nombres será el marco asociado al ámbito de la ligadura.

## 2.3. Ámbito de un identificador

A veces, por economía del lenguaje, se suele hablar del «**ámbito de un identificador**», en lugar de hablar del «*ámbito de la ligadura que liga ese identificador con un valor*».

Por ejemplo, en el siguiente *script*:

```
x = 25
```

tenemos que:

- En el ámbito global, hay una definición que liga al identificador `x` con el valor `25`.
- Por tanto, se dice que **el ámbito de esa ligadura es el ámbito global**.
- Pero también se suele decir que «*el identificador `x` es global*» o, simplemente, que «*`x` es global*».

O sea, se **asocia al ámbito** no la ligadura, sino **el identificador en sí**.

Pero hay que tener cuidado, ya que ese mismo identificador puede aparecer en ámbitos diferentes y, por tanto, ligarse en ámbitos diferentes.

Así que no tendría sentido hablar del ámbito que tiene ese identificador (ya que podría tener varios) sino, más bien, **del ámbito que tiene una aparición concreta de ese identificador**.

Por eso, sólo deberíamos hablar del ámbito de un identificador cuando no haya ninguna ambigüedad respecto a qué aparición concreta nos estamos refiriendo.

Por ejemplo, en el siguiente *script*:

```
1 x = 4
2 suma = (lambda x, y: x + y)(2, 3)
```

el identificador `x` que aparece en la línea 1 y el identificador `x` que aparece en la línea 2 pertenecen a ámbitos distintos (como veremos en breve) aunque sea el mismo identificador.

## 2.4. Ámbito de un parámetro

**El cuerpo de la expresión lambda determina un ámbito.**

Por ejemplo, supongamos la siguiente llamada a una expresión lambda:

```
(lambda x, y: x + y)(2, 3)
```

**Al llamar a la expresión lambda** (es decir, al aplicar la expresión lambda a unos argumentos), se empieza a ejecutar su cuerpo y, por tanto, **se entra en dicho ámbito**.

En ese momento, **se crea un nuevo marco** en la memoria, que representa esa ejecución concreta de dicha expresión lambda.

Lo que ocurre justo a continuación es que **cada parámetro de la expresión lambda se liga a uno de los argumentos** en el orden en que aparecen en la llamada a la expresión lambda (primer parámetro con primer argumento, segundo con segundo, etcétera).

En el ejemplo anterior, es como si el intérprete ejecutara las siguientes definiciones dentro del ámbito de la expresión lambda:

```
x = 2
y = 3
```

Las ligaduras que crean esas definiciones **se almacenan en el marco de la llamada a la expresión lambda**.

Ese marco se eliminará de la memoria al salir del ámbito de la expresión lambda, es decir, cuando se termine de ejecutar el cuerpo de la expresión lambda al finalizar la llamada a la misma.

Por tanto, las ligaduras se destruyen de la memoria al eliminarse el marco que las almacena.

La próxima vez que se llame a la expresión lambda, se volverán a ligar sus parámetros con los argumentos que haya en esa llamada.

Por ejemplo, supongamos que tenemos esta situación:

```
suma = lambda x, y: x + y
a = suma(4, 3)
b = suma(8, 9)
```

En la primera llamada, se entrará en el ámbito determinado por el cuerpo la expresión lambda, se creará el marco que representa a esa llamada, y se ejecutarán las siguientes definiciones dentro del ámbito:

```
x = 4
y = 3
```

lo que creará las correspondientes ligaduras y las almacenará en el marco de esa llamada.

Después, evaluará el cuerpo de la expresión lambda y devolverá el resultado, saliendo del cuerpo de la expresión lambda y, por tanto, del ámbito que determina dicho cuerpo, lo que hará que se destruya el marco y, en consecuencia, las ligaduras que contiene.

En la siguiente llamada ocurrirá lo mismo pero, esta vez, las definiciones que se ejecutarán serán las siguientes:

```
x = 8
y = 9
```

lo que creará otras ligaduras, que serán destruidas luego cuando se destruya el marco que las contiene, al finalizar la ejecución del cuerpo de la expresión lambda.

Es importante hacer notar que **en ningún momento se está haciendo un rebinding de los parámetros**, ya que cada vez que se llama de nuevo a la expresión lambda, se está creando una ligadura nueva sobre un identificador que no estaba ligado.

En consecuencia, podemos decir que:

- El **ámbito de la ligadura** entre un parámetro y su argumento es el **cuerpo** de la expresión lambda, así que la **visibilidad** del parámetro (y de la ligadura) es ese cuerpo.
- Esa ligadura se crea justo después de entrar en ese ámbito, así que se puede acceder a ella en cualquier parte del cuerpo de la expresión lambda, por lo que su **tiempo de vida** va desde el principio hasta el final de la llamada.
- El **espacio de nombres** que almacena las ligaduras entre parámetros y argumentos es el **marco** que se crea al llamar a la expresión lambda.

Esto se resume diciendo que «el **ámbito de un parámetro** es el **cuerpo** de su expresión lambda».

También se dice que el parámetro tiene un **ámbito local** y un **almacenamiento local** al cuerpo de la expresión lambda.

Resumiendo: el parámetro es **local** a dicha expresión lambda.

Por tanto, **sólo podemos acceder al valor de un parámetro dentro del cuerpo de su expresión lambda**.

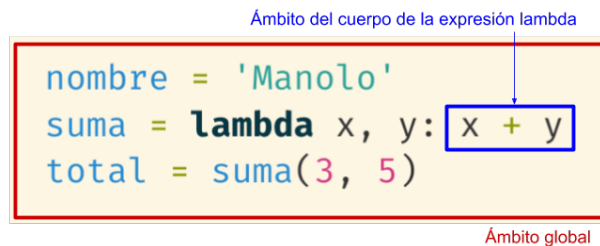
Por ejemplo, en el siguiente código:



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

el cuerpo de la expresión lambda ligada a `suma` determina su propio ámbito.

Por tanto, en el siguiente código tenemos dos ámbitos: el ámbito global (más externo) y el ámbito del cuerpo de la expresión lambda (más interno y anidado dentro del ámbito global):



Además, cada vez que se llama a `suma`, la ejecución del programa entra en su cuerpo, lo que crea un nuevo marco que almacena las ligaduras entre sus parámetros y los argumentos usados en esa llamada.

#### En resumen:

El **ámbito de un parámetro** es el ámbito de la ligadura que se establece entre éste y su argumento correspondiente, y se corresponde con el **cuerpo** de la expresión lambda donde aparece.

Por tanto, el parámetro sólo existe dentro del cuerpo de la expresión lambda y no podemos **acceder** a su valor fuera del mismo; por eso se dice que tiene un **ámbito local** a la expresión lambda.

Además, la **ligadura** entre el parámetro y su argumento **se almacena en el marco** de la llamada a la expresión lambda, y por eso se dice que tiene un **almacenamiento local** a la expresión lambda.

Los ámbitos léxicos permiten ligaduras locales a ciertas construcciones sintácticas, lo cual nos permite programar definiendo partes suficientemente independientes entre sí.

Esto es la base de la llamada *programación modular*.

Por ejemplo, nos permite crear funciones sin preocuparnos de si los nombres de los parámetros ya han sido utilizados en otras partes del programa.

Igualmente, nos permite crear programas sin preocuparnos de si estamos usando nombres que ya han sido usadas en el interior de alguna de las funciones del programa.

De lo contrario, se podría provocar lo que se conoce como **name clash** (*conflicto de nombres* o *choque de nombres*), que es el problema que se produce cuando usamos el mismo nombre para varias cosas diferentes y que impide que se puedan acceder a todas al mismo tiempo.

Lo que impide el *name clash* son dos cosas:

- Los *ámbitos* hacen que los nombres sólo sean visibles en ciertas zonas.
- Los *espacios de nombres* permiten que un mismo nombre pueda ligarse a diferentes nombres simultáneamente.

## 2.5. Ámbito de un identificador cuantificado

Hemos visto que a los **parámetros** de una expresión lambda se les llama **identificadores cuantificados** cuando aparecen dentro del cuerpo de dicha expresión lambda.

Por tanto, todo lo que se dijo sobre el ámbito de un parámetro se aplica exactamente igual al ámbito de un identificador cuantificado.

Recordemos que el ámbito de un parámetro es el cuerpo de su expresión lambda, que es la porción de código donde podemos acceder al valor del argumento con el que está ligado.

Por tanto, **el ámbito de un identificador cuantificado es el cuerpo de la expresión lambda** donde aparece, y es el único lugar dentro del cual podremos acceder al valor del identificador cuantificado (que también será el valor del argumento con el que está ligada).

Por eso también se dice que el identificador cuantificado tiene un **ámbito local** al cuerpo de la expresión lambda.

### En resumen:

El **ámbito de un identificador cuantificado** es el ámbito de la ligadura que se crea entre ésto y su argumento correspondiente, y se corresponde con el **cuerpo** de la expresión lambda donde aparece.

Por tanto, el identificador cuantificado sólo existe dentro del cuerpo de la expresión lambda y no podemos **acceder** a su valor fuera del mismo; por eso se dice que tiene un **ámbito local** a la expresión lambda.

Además, **la ligadura** entre el identificador cuantificado y su argumento **se almacena en el marco** de la llamada a la expresión lambda, y por eso se dice que tiene un **almacenamiento local** a la expresión lambda.

O sea: con los **identificadores cuantificados** ocurre exactamente lo mismo que con los **parámetros**, ya que, de hecho, **un parámetro y un identificador cuantificado son la misma cosa**, como ya hemos visto.

### Ejemplo

En el siguiente *script*:

```
1 # Aquí empieza el script (no hay más definiciones antes de esta línea):
2 producto = lambda x: x * x
3 y = producto(3)
4 z = x + 1 # da error
```

Hay dos ámbitos: el ámbito global y el ámbito local definido por el cuerpo de la expresión lambda (o sea, la expresión `x * x`).

Esa expresión lambda tiene un parámetro (`x`) que aparece como el identificador cuantificado `x` en el cuerpo de la expresión lambda.

El ámbito del parámetro `x` (o, lo que es lo mismo, el identificador cuantificado `x`) es el **cuerpo** de la expresión lambda.

Por tanto, fuera de ese cuerpo, no es posible acceder al valor del identificador cuantificado  $x$ , al encontrarnos **fuera de su ámbito** (la ligadura **sólo es visible dentro del cuerpo** de la expresión lambda).

Por eso, la línea 4 dará un error al intentar acceder al valor  $x$ , cuya ligadura no es visible fuera de la expresión lambda.

## 2.6. Ámbito de un identificador libre

Los identificadores y ligaduras que no tienen ámbito local se dice que tienen un **ámbito no local** o, a veces, un **ámbito más global**.

Si, además, ese ámbito resulta ser el **ámbito global**, decimos directamente que esos identificadores o ligaduras son **globales**.

Por ejemplo, los **identificadores libres** que aparecen en una expresión lambda no son locales a dicha expresión (ya que no representan parámetros de la expresión) y, por tanto:

1. Tienen un ámbito más global que el cuerpo de dicha expresión lambda.
2. Se almacenarán en otro espacio de nombres distinto al marco que se crea al llamar a la expresión lambda.

## 3. Funciones recursivas

### 3.1. 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

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

## 3.2. 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**.

## 3.3. 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  $n$ :

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

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:

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

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

$$\left\{ \begin{array}{l} \text{Pre : } n \geq 0 \\ \text{factorial}(n:\text{int}) \rightarrow \text{int} \\ \text{Post : } \text{factorial}(n) = n! \end{array} \right.$$

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

```
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.

### 3.4. 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

#### 3.4.1. 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:

$$\text{fact}(0) = 1$$

### 3.4.2. 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.

### 3.4.3. 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:

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

### 3.5. 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*:

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

#### 3.5.1. 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**.

### 3.5.2. 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:

$$\begin{aligned} acumulador_{nuevo} &= acumulador_{viejo} \cdot contador_{viejo} \\ contador_{nuevo} &= contador_{viejo} - 1 \end{aligned}$$

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 $\underline{n}$	Proporcional a $\underline{n}$
Iterativo	Proporcional a $\underline{n}$	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.

### 3.6. Recursividad múltiple

Una función tiene **recursividad múltiple** cuando la misma llamada a la función recursiva 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:

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

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

$$\left\{ \begin{array}{l} \text{Pre : } n \geq 0 \\ \text{fib}(n: \text{int}) \rightarrow \text{int} \\ \text{Post : } \text{fib}(n) = \text{el } n\text{-ésimo término de la sucesión de Fibonacci} \end{array} \right.$$

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:

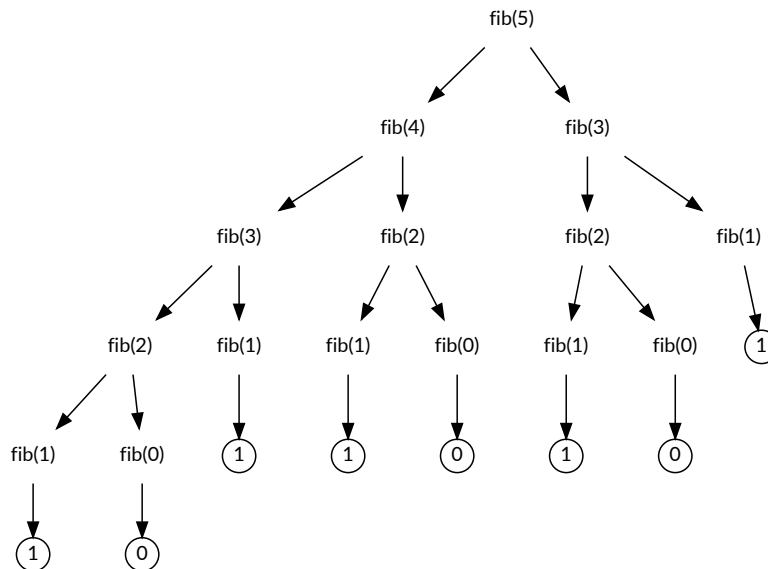
```
fib = lambda n: 0 if n == 0 else \
    1 if n == 1 else \
    fib(n - 1) + fib(n - 2)
```

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  $\text{fib}(n)$  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:

$$\begin{aligned} a_{\text{nuevo}} &= b_{\text{viejo}} \\ b_{\text{nuevo}} &= b_{\text{viejo}} + a_{\text{viejo}} \end{aligned}$$

Después de  $n$  pasos,  $a$  y  $b$  contendrán  $\text{fib}(n)$  y  $\text{fib}(n + 1)$ , 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.

### 3.7. 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.

## 4. Abstracciones funcionales

### 4.1. Las funciones como abstracciones y generalizaciones

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.

«*Abstraer*» es centrarse en lo importante en un determinado momento e ignorar lo que en ese momento no resulta importante.

«*Crear una abstracción*» es meter un mecanismo más o menos complejo dentro de una caja negra y darle un nombre, de forma que podamos referirnos a todo el conjunto simplemente usando su nombre y 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 *lo que hace*, sin necesidad de saber *cómo lo hace* ni de qué elementos está formada *internamente*.

La **abstracción** es el principal instrumento 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.

Las **funciones son**, por tanto, **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 * x ** 2
```

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

Los detalles de implementación quedan ocultos y por eso también decimos que **cubo** es una **abstracción**.

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

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.

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 *eleva al cubo*.**

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 (*eleva al cubo*).

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

Por tanto, algunas veces, analizando ciertos *casos particulares*, observamos que se repite el mismo

patrón en todos ellos, y de ahí extraemos un *caso general* que agrupa a todos los posibles casos particulares que cumplen el mismo patrón (es decir, hacemos una **generalización**).

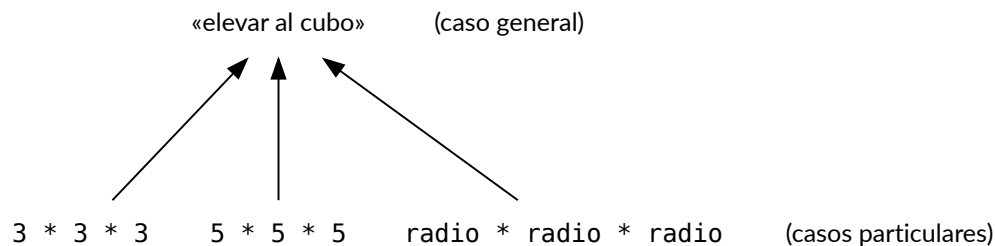
A ese caso general le damos un nombre y ocultamos sus detalles internos en una «caja negra», y con esto creamos una **abstracción**.

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 «elear al cubo».
- Cuando creamos **casos generales a partir de patrones que se repiten** en varios casos particulares.

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 «elear 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 **abstracción** (porque se puede invocar usando su nombre sin necesidad de conocer su interior), 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.

El proceso conceptual detallado sería el siguiente:

1. Partimos de casos particulares que comparten un patrón que se repite:

$3 * 3 * 3$   
 $5 * 5 * 5$



En este caso, el patrón es el de «algo» que se multiplica por sí mismo tres veces.

En todos los casos particulares de ese patrón repetido vemos que hay elementos que se son iguales y otros que son diferentes.

Por ejemplo, los  $*$  son iguales y lo que varía es el «objeto» que se multiplica.

2. Generalizamos estos casos particulares, convirtiendo en identificadores libres aquellas partes en las que los casos particulares se diferencian, es decir, las partes que no son comunes:

```
x * x * x
```

El valor de esta expresión actualmente depende del valor al que esté ligado el identificador  $x$ , y decimos que ese identificador es *libre*.

Los **identificadores libres** son nombres que no se sabe de antemano a qué valores van a estar ligados, ya que dependen del *contexto*, es decir, de lo que hay fuera de la expresión cuando se va a evaluar. En todo caso, un identificador libre estará ligado siempre a un mismo valor y, en ese sentido, se pueden considerar constantes.

Se dice que una expresión con identificadores libres está *abierta*, porque su valor depende de elementos externos a ella.

3. Abstraemos dándole un nombre a toda la expresión, de forma que ahora podríamos usar el nombre en lugar de la expresión:

```
cubo = x * x * x
```

El valor de `cubo` sigue dependiendo del valor ligado a  $x$ , ya que esta  $x$  es libre.

Por tanto, para deducir qué valor tendrá `cubo` tendremos que seguir conociendo su interior: tenemos que saber que su valor se calcula a partir del valor que tiene la variable libre  $x$ .

Esto hace que `cubo` no sea una buena abstracción, ya que no funciona como una *caja negra*. Esto se debe a que la abstracción está *abierta*.

Cuando una parte de un programa está *abierta* resulta más difícil de programar y de razonar sobre ella, ya que su comportamiento depende del resto del programa. Lo que nos interesa (siempre que sea posible) es que la abstracción esté *cerrada*.

4. Generalizamos aún más, *parametrizando* los identificadores libres utilizando el cuantificador **lambda**, creando así una **abstracción lambda**:

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

Un **cuantificador** es un símbolo que *cierra* expresiones convirtiendo constantes en variables. En este caso, las constantes 3 o 5 (o cualquier otra posible, como la  $x$  libre) se han convertido en un parámetro.

Los **parámetros** son nombres cuyo valor cambia dependiendo de los argumentos de la llamada. Por eso se pueden considerar *variables* en el sentido matemático del término.

Ahora hemos *cerrado* la expresión creando una función en la que los identificadores libres ya no son libres sino *parámetros* de la expresión lambda, así que la expresión ya no depende de nada

que haya en el exterior de la misma.

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

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

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

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.

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* se implementa.

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.

## 4.2. 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:

$$\left\{ \begin{array}{l} \text{Entrada : } n \in \mathbb{R} \\ \text{cuadrado} \\ \text{Salida : } n^2 \end{array} \right.$$

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* (programas que forman parte de otros programas más grandes).

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

$$\left\{ \begin{array}{l} \text{Pre : } \text{True} \\ \text{cuadrado}(n: \text{float}) \rightarrow \text{float} \\ \text{Post : } \text{cuadrado}(n) = n^2 \end{array} \right.$$

«**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`:

$$\left\{ \begin{array}{l} \text{Pre : } \text{car} \neq "" \wedge \text{len}(\text{car}) = 1 \\ \qquad \text{cuenta}(\text{cadena: str}, \text{car: str}) \rightarrow \text{int} \\ \text{Post : } \text{cuenta}(\text{cadena}, \text{car}) \geq 0 \wedge \\ \qquad \text{cuenta}(\text{cadena}, \text{car}) = \text{cadena.count}(\text{car}) \end{array} \right.$$

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:

$$\left\{ \begin{array}{l} \text{Pre : } \text{len}(\text{car}) = 1 \\ \qquad \text{cuenta}(\text{cadena: str}, \text{car: str}) \rightarrow \text{int} \\ \text{Post : } \text{cuenta}(\text{cadena}, \text{car}) = \text{cadena.count}(\text{car}) \end{array} \right.$$

## Ejercicio

### 2. ¿Por qué?

Finalmente, podríamos escribir la misma especificación en lenguaje natural:

**Pre :** *car* debe ser un único carácter  
`cuenta(cadena: str, car: str) -> int`  
**Post :** `cuenta(cadena, car)` devuelve el número de veces  
que aparece el carácter *car* en la cadena *cadena*.  
Si *cadena* es vacía o *car* no aparece nunca en la  
cadena *cadena*, debe devolver 0.

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.

## 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.