

Abstracciones funcionales

Ricardo Pérez López

IES Doñana, curso 2024/2025

Generado el 2025/07/09 a las 01:04:00

Índice

1. Abstracciones lambda	2
1.1. Expresiones lambda	2
1.2. Parámetros y cuerpos	2
1.3. Aplicación funcional	2
1.3.1. Evaluación de una aplicación funcional	3
1.3.2. Funciones con nombre	4
1.4. Identificadores libres de una expresión lambda	5
2. Ámbitos	6
2.1. Ámbitos léxicos	6
2.2. Ámbito de una definición y de una ligadura	7
2.2.1. Almacenamiento	8
2.2.2. Visibilidad	9
2.2.3. Tiempo de vida	9
2.3. Ámbito de un identificador	12
2.4. Ámbito de un parámetro	12
2.5. Ámbito de un identificador libre	15
3. Abstracciones funcionales	15
3.1. Pureza	15
3.2. Las funciones como abstracciones	16
3.2.1. Especificaciones de funciones	19

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

```

1.4. Identificadores libres de una expresión lambda

Si un *identificador* aparece en el *cuerpo* de una expresión lambda pero no aparece también en la *lista de parámetros* de esa expresión lambda, decimos que es un **identificador libre** en 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 ninguno de esos identificadores son libres.

En cambio, en la expresión lambda:

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

z es un identificador libre, ya que no aparece en la lista de parámetros, donde sí aparecen *x* e *y*.

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 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 definen 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.

En un momento dado, el **ámbito actual** es el ámbito más interno en el que se encuentra la instrucción que se está ejecutando actualmente (es decir, la **instrucción actual**).

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).

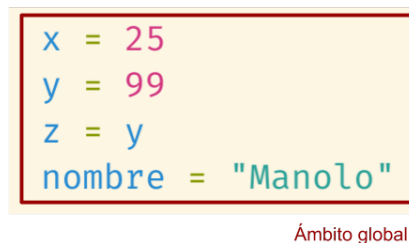
En la práctica, es lo mismo hablar del «ámbito de una definición» que del «ámbito de la ligadura que se creará al ejecutar la definición», ya que son la misma cosa.

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.

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



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

Ámbito global

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 que 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 *atributo* del objeto con un valor.

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`.

2. Si la ligadura no se crea dentro de un objeto usando el operador punto (`.`), entonces el espacio de nombres dependerá del ámbito:
 - a. Si el ámbito donde se crea la ligadura lleva asociado un espacio de nombres, ese espacio de nombres almacenará las ligaduras que se crean dentro de ese ámbito.
 - b. Si no, entonces la ligadura se almacenará en el espacio de nombres del ámbito más interno que contenga al actual y que sí lleve asociado un espacio de nombres.

Por tanto, a la hora de almacenar una ligadura, se van mirando todos los ámbitos desde el ámbito actual, pasando por todos los ámbitos que incluyen a éste (en orden, del más interno al más externo), hasta encontrar el primer ámbito que lleve asociado un espacio de nombres.

En cualquier caso, aquí el espacio de nombres seleccionado siempre será un marco.

Cuando la ligadura se almacena en el marco global, se dice que tiene **almacenamiento global**.

2.2.2. Visibilidad

La visibilidad de una ligadura indica cuándo y de qué manera es visible esa ligadura.

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 en el ámbito donde se definió la ligadura, el cual va asociado al marco donde se almacena 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.

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 donde 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 dejará de existir allí donde termine su ámbito.

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 y el tiempo de vida de las ligaduras vendrán determinadas 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` \rightarrow `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:

```
>>> 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
```

En resumen:

- Para poder acceder a un atributo de un objeto debemos indicar éste 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).

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:

- a. Si es un atributo de un objeto, la visibilidad será el objeto que contiene la ligadura.
- b. En caso contrario, la visibilidad será 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:

- a. 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).
- b. 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.

En Python:

- a. Si el identificador ligado es un atributo de un objeto, el espacio de nombres será el objeto que lo contiene.

b. 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 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.

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.
- 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):

```
nombre = 'Manolo'
suma = lambda x, y: x + y
total = suma(3, 5)
```

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 (que es la base de la *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.

2.5. 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, tienen un ámbito más global que el cuerpo de dicha expresión lambda y se almacenarán en otro espacio de nombres distinto al marco que se crea al llamar a la expresión lambda.

3. Abstracciones funcionales

3.1. 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 más 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 ejemplo anterior, 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* en el entorno:

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

En este otro ejemplo, escribimos 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` en el entorno actual.

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.

3.2. 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.

«*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 $3*3*3$, $y*y*y$, 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.

A ese caso general le damos un nombre y ocultamos sus detalles internos en una «caja negra».

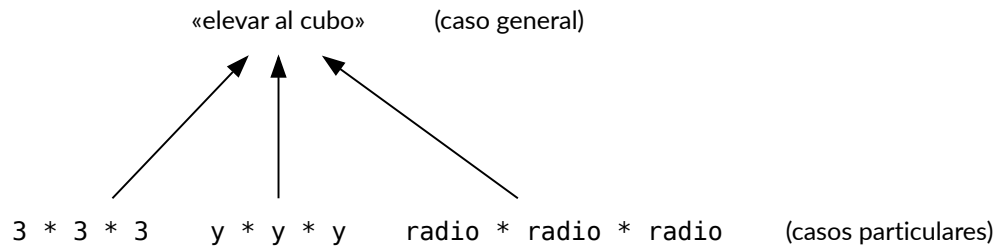
Eso es una **abstracción**.

En resumen, creamos abstracciones:

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

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 «*eleva al cubo*»:



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.

3.2.1. Especificaciones de funciones

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**:

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

«**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 signature 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**:

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

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:

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

Ejercicio

1. ¿Por qué?

Finalmente, podríamos escribir la misma especificación en lenguaje natural:

$$\left\{ \begin{array}{l} \text{Pre : } \text{car debe ser un único carácter} \\ \qquad \text{cuenta}(\text{cadena}:\text{str}, \text{car}:\text{str}) \rightarrow \text{int} \\ \text{Post : } \text{cuenta}(\text{cadena}, \text{car}) \text{ devuelve el número de veces} \\ \qquad \text{que aparece el carácter } \text{car} \text{ en la cadena } \text{cadena}. \\ \qquad \text{Si } \text{cadena} \text{ es vacía o } \text{car} \text{ no aparece nunca en la} \\ \qquad \text{cadena } \text{cadena}, \text{ debe devolver } 0. \end{array} \right.$$

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.