

Programación funcional (II)

Ricardo Pérez López

IES Doñana, curso 2020/2021

Generado el 10 de noviembre de 2020 a las 19:01:00

Índice general

1. Abstracciones lambda	2
1.1. Expresiones lambda	2
1.2. Parámetros y cuerpos	3
1.3. Aplicación funcional	3
1.3.1. Evaluación de una aplicación funcional	3
1.3.2. Llamadas a funciones	4
1.4. Variables ligadas y libres	5
2. Ámbitos léxicos	6
2.1. Ámbitos	6
2.2. Ámbito de una ligadura y de creación de una ligadura	7
2.3. Ámbito de un identificador	8
2.4. Ámbito de un parámetro	9
2.5. Ámbito de una variable ligada	9
2.6. Entorno (<i>environment</i>)	10
2.7. Ámbitos, marcos y entornos	11
2.8. Ligaduras <i>sombreadas</i>	13
2.9. Renombrado de parámetros	14
3. Evaluación	15
3.1. Evaluación de expresiones con entornos	15
3.2. Evaluación de expresiones lambda con entornos	17
3.2.1. Visualización en <i>Pythontutor</i>	21
3.3. Estrategias de evaluación	22
3.3.1. Orden de evaluación	22
3.3.2. Composición de funciones	24
3.3.3. Evaluación estricta y no estricta	26
4. Abstracciones funcionales	27
4.1. Pureza	27
4.2. Las funciones como abstracciones	29
4.2.1. Especificaciones de funciones	31

5. Computabilidad	34
5.1. Funciones y procesos	34
5.2. Funciones <i>ad-hoc</i>	34
5.3. Funciones recursivas	35
5.3.1. Definición	35
5.3.2. Casos base y casos recursivos	36
5.3.3. El factorial	37
5.3.4. Diseño de funciones recursivas	38
5.3.5. Recursividad lineal	39
5.3.6. Recursividad múltiple	42
5.3.7. Recursividad final y no final	44
5.4. La pila de control	45
5.5. Un lenguaje Turing-completo	48
6. Tipos de datos recursivos	48
6.1. Cadenas	48
6.2. Tuplas	48
6.3. Rangos	49
6.4. Conversión a tupla	51
7. Funciones de orden superior	51
7.1. Concepto	52
7.2. <code>map</code>	54
7.3. <code>filter</code>	56
7.4. <code>reduce</code>	56
7.5. Expresiones generadoras	57

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:

```
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 **+**).

1.3. Aplicación funcional

De la misma manera que decíamos 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)** que denota el valor **cinco** (o sea, que la llamada a la función devuelve **5**).

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)
```

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* (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`.

1.3.2. Llamadas a funciones

Si hacemos la siguiente definición:

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

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)` implicaría realizar los siguientes tres pasos y en este orden:

1. Sustituir el nombre de la función `suma` por su definición.
2. Evaluar sus argumentos.
3. Aplicar la expresión lambda a sus argumentos.

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, 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)           # evalúa 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. Variables ligadas y libres

Si un *identificador* que aparece en el *cuerpo* de una expresión lambda, también aparece en la *lista de parámetros* de esa expresión lambda, a ese identificador le llamamos **variable ligada** de la expresión lambda.

En caso contrario, le llamamos **variable 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**) son variables ligadas, ya que ambos aparecen también en la lista de parámetros de la expresión lambda.

En cambio, en la expresión lambda:

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

x e **y** son variables ligadas mientras que **z** es una variable libre.

En realidad, una **variable ligada** y un **parámetro** son la misma cosa.

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

- Si aparece **antes** del «:», le llamamos «*parámetro*».

- Si aparece **después** del «:», le llamamos «*variable ligada*».

Por ejemplo: en la siguiente expresión lambda:

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

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

2. Ámbitos léxicos

2.1. Ambitos

Existen ciertos bloques sintácticos que, cuando se ejecutan, provocan la creación de nuevos marcos.

Cuando eso ocurre, decimos que ese bloque sintáctico define un **ámbito**, y ese ámbito viene definido por la porción del código fuente que ocupa ese bloque sintáctico dentro del programa, de forma que:

- Cuando la ejecución del programa **entra** en el ámbito, se **crea** un nuevo marco.
- Cuando la ejecución se **sale** del ámbito, se **destruye** su marco.

Cada marco va asociado con un ámbito, y cada ámbito tiene su marco.

Los ámbitos **se anidan recursivamente**, o sea, que están contenidos unos dentro de otros, así que una instrucción puede estar en varios ámbitos al mismo tiempo (anidados unos dentro de otros).

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

El concepto de *ámbito* es un concepto nada trivial y, a medida que vayamos incorporando nuevos elementos al lenguaje, tendremos que ir adaptándolo para tener en cuenta más condicionantes.

Por ahora sólo tenemos un ámbito llamado **ámbito global**:

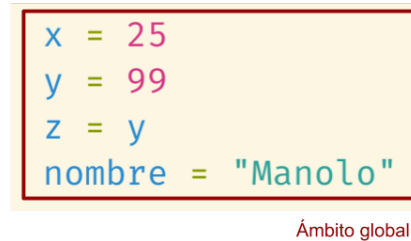
- Si se está ejecutando un *script* en el intérprete por lotes (con `python script.py`), el *ámbito global* abarca todo el *script*, desde la primera instrucción hasta la última.
- Si estamos en el intérprete interactivo (con `python` o `ipython3`), el *ámbito global* abarca toda nuestra sesión con el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

En el momento en que se empieza a ejecutar un *script* o se arranca una nueva sesión con el intérprete interactivo, se entra en el *ámbito global*, lo que provoca la creación de un nuevo marco llamado **marco global**.

Del ámbito global sólo se sale cuando se finaliza la ejecución del *script* o se cierra el intérprete interactivo.

Las ligaduras que se crean dentro de un ámbito se almacenan en el marco asociado a ese ámbito.

Por ejemplo, en el siguiente *script* se realizan cuatro definiciones. Todas ellas se ejecutan en el ámbito global, que es el único ámbito que existe en el *script*:



```

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

```

Ámbito global

Ninguna de esas instrucciones crea un nuevo ámbito, por lo que todas se ejecutan en el ámbito actual (el ámbito global) y todas las ligaduras que se crean se almacenan en el marco global.

2.2. Ámbito de una ligadura y de creación de una ligadura

El **ámbito de una ligadura** es la porción del código fuente en la que existe dicha ligadura.

El **ámbito de creación de una ligadura** es el ámbito más interno donde se crea una ligadura, y determina el **marco** donde se almacenará la ligadura.

El ámbito de una ligadura no tiene por qué coincidir exactamente con su ámbito de creación.

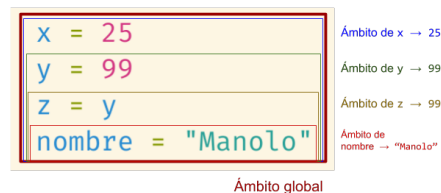
Esto es así porque una ligadura empieza a existir en el momento en el que se ejecuta la instrucción que la crea, y no antes.

Por tanto, si en un momento dado se está ejecutando una instrucción dentro de un ámbito, las ligaduras que existirán dentro de ese ámbito en ese momento serán sólo las que se hayan creado en ese ámbito **hasta ese momento**.

En consecuencia, el ámbito de una ligadura:

- Empieza en el punto donde se crea la ligadura.
- Termina donde lo hace el *ámbito de creación* de la ligadura.

En el siguiente ejemplo vemos los ámbitos de cada ligadura. Todas esas ligaduras se definen en el ámbito global, por lo que el **ámbito de creación** de todas ellas es el **ámbito global**.



Es importante no confundir «**ámbito**», «**ámbito de creación de una ligadura**» y «**ámbito de una ligadura**».

Ámbito:

Porción de código de un programa que crea un marco al entrar y lo destruye al salir. Su marco almacena las ligaduras que se crean dentro de ese ámbito.

Ámbito de creación de una ligadura:

Ámbito más interno en el que se crea una ligadura y que determina el marco donde se almacenará la ligadura.

Ámbito de una ligadura:

Porción de código en el que la ligadura existe, que va desde su creación hasta el final del ámbito más interno que la contiene (su *ámbito de creación*). No es un ámbito como tal y, por tanto, no crea marcos.

La creación de una ligadura (por ejemplo, con una sentencia de definición) no define un nuevo ámbito y, por tanto, no crea un nuevo marco.

Por eso, las ligaduras se almacenan en el marco del ámbito donde se crean (es el **ámbito de creación de la ligadura**).

Hasta ahora, todas las ligaduras las hemos definido en el ámbito global, por lo que se almacenan en el marco global.

Por eso también decimos que esas ligaduras tienen ámbito global, o que pertenecen al ámbito global, o que están definidas en el ámbito global, o que son **globales**.

Ampliaremos ahora el concepto de *ámbito* para incluir los aspectos nuevos que incorporan las expresiones lambda.

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 creación de la ligadura que liga ese identificador con un valor*».

Por ejemplo, en el siguiente *script*:

```
x = 25
```

tenemos que:

- En el ámbito global, se crea una ligadura que liga al identificador `x` con el valor `25`.
- Por tanto, se dice que el *ámbito de creación de esa ligadura* es el ámbito global.
- Pero también se suele decir que «*el identificador `x` es global*» (o, simplemente, que «*`x` es global*»), **asociando al ámbito** no la ligadura, sino **el identificador en sí**.

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

Por tanto, 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 que aparece en la línea 2 pertenecen a ámbitos distintos (como veremos en breve).

2.4. Ámbito de un parámetro

El cuerpo de la expresión lambda define un ámbito.

Por tanto, cuando se va a evaluar una aplicación funcional, se entra en dicho ámbito y eso crea un marco.

Cuando se aplica una expresión lambda a unos argumentos, **cada parámetro de la expresión lambda se liga a uno de esos argumentos** en el orden en que aparecen en la aplicación funcional (primer parámetro con primer argumento, segundo con segundo, etcétera).

Esas ligaduras se crean justo al entrar en el ámbito que define el cuerpo de la expresión lambda.

Por tanto, **se almacenan en el marco de la expresión lambda** nada más entrar en el cuerpo de la expresión lambda.

En consecuencia, podemos decir que:

- El **ámbito de creación de la ligadura** entre un parámetro y su argumento es el **cuerpo** de la expresión lambda.
- El **ámbito de esa ligadura** coincide con su ámbito de creación.

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** al cuerpo de la expresión lambda o que es **local** a dicha expresión lambda.

Como el ámbito de una ligadura es la porción del código en el que dicha ligadura tiene validez, eso significa que **sólo podemos acceder al valor de un parámetro dentro del cuerpo de su expresión lambda**.

En resumen:

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

2.5. Ámbito de una variable ligada

Hemos visto que a los **parámetros** de una expresión lambda se les llama **variables ligadas** 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 una variable ligada.

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 una variable ligada también es el cuerpo de la expresión lambda** donde aparece, y es el único lugar dentro del cual podremos acceder al valor de la variable ligada (que también será el valor del argumento con el que está ligada).

En consecuencia, también se dice que la variable ligada tiene un **ámbito local** al cuerpo de la expresión lambda o que es **local** a dicha expresión lambda.

Por contraste, las variables, 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 esa variable, identificador o ligadura es **global**.

Por ejemplo, las **variables 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.

En resumen:

El **ámbito de una variable ligada** es el ámbito de la ligadura que se establece entre ésta y su argumento correspondiente, y coincide con el **cuerpo** de la expresión lambda donde aparece.

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: (1) el ámbito global y (2) el ámbito local definido el cuerpo de la expresión lambda (la expresión `x * x`).

La expresión lambda de la línea 2 tiene un parámetro (`x`) que aparece como la variable ligada `x` en el cuerpo de la expresión lambda.

El ámbito de la variable ligada `x` es el **cuerpo** de la expresión lambda.

Por tanto, fuera del cuerpo de la expresión lambda, no es posible acceder al valor de la variable ligada `x`, al encontrarnos **fuera de su ámbito** (la `x` sólo está ligada en el cuerpo de la expresión lambda).

Por eso, la línea 4 dará un error al intentar acceder al valor del identificador `x`, que no está ligado en el ámbito actual (el global).

2.6. Entorno (*environment*)

El **entorno** es una extensión del concepto de *marco*.

Durante la ejecución del programa, se van creando y destruyendo marcos a medida que la ejecución va entrando y saliendo de ámbitos.

Como los ámbitos están anidados unos dentro de otros, una instrucción puede estar en varios ámbitos al mismo tiempo.

Eso hace que, en un momento dado, pueda haber varios marcos activos en memoria al mismo tiempo (uno por cada ámbito en el que se encuentre la instrucción que se está ejecutando actualmente).

Según se van creando en memoria, esos marcos van enlazándose unos con otros creando una **secuencia de marcos** que se denomina **entorno** (del inglés, *environment*).

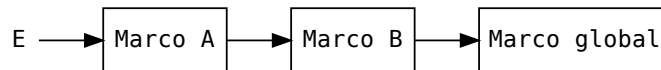
El entorno nos dice **todos los identificadores que son accesibles en un momento concreto de la ejecución del programa, y con qué valores están ligados**.

En un momento dado, el entorno contendrá más o menos marcos dependiendo de por dónde haya pasado la ejecución del programa hasta ese momento.

El entorno, por tanto, es un concepto **dinámico** que **depende del momento en el que se calcule**, es decir, de por dónde va la ejecución del programa (o, lo que es lo mismo, de qué instrucciones se han ejecutado hasta ahora y en qué ámbitos se ha entrado).

El entorno **siempre contendrá**, al menos, un marco: el *marco global*.

El marco global siempre será el último de la secuencia de marcos que forman el entorno.



2.7. Ámbitos, marcos y entornos

Recordemos que un marco es un conjunto de ligaduras.

Y que un entorno es una secuencia de marcos que contienen todas las ligaduras válidas en un punto concreto de la ejecución del programa.

Cuando la ejecución del programa entra dentro de un ámbito, **se crea un nuevo marco asociado a ese ámbito**.

Además, **el cuerpo de una expresión lambda define un nuevo ámbito**.

Por tanto, cuando se aplica una expresión lambda a unos argumentos, **se crea un nuevo marco que contiene las ligaduras que ligan a los parámetros con los valores de esos argumentos**.

Ese nuevo marco se enlaza con el marco del ámbito que lo contiene (el marco del ámbito más interno *apunta* al del más externo), de manera que el último marco de la secuencia siempre es el marco global.

El marco desaparece cuando el flujo de control del programa se sale del ámbito, ya que cada marco va asociado a un ámbito.

Se va formando así una secuencia de marcos que representa el **entorno** del programa en un punto dado del mismo.

El **ámbito** es un concepto *estático*: es algo que existe y se reconoce simplemente leyendo el código del programa, sin tener que ejecutarlo.

El **marco** es un concepto *dinámico*: es algo que se crea y se destruye a medida que vamos entrando o saliendo de un ámbito, y contiene las ligaduras que se definen dentro de ese ámbito.

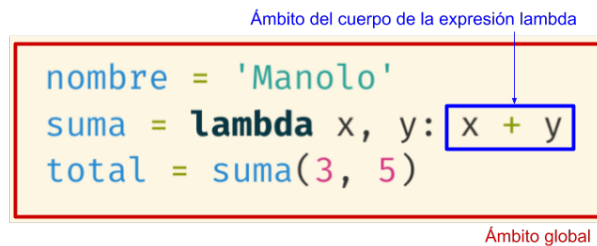
A partir de ahora ya no vamos a tener un único marco (el *marco global*) sino que tendremos, además, al menos uno más cada vez que se aplique una expresión lambda a unos argumentos.

Por ejemplo:

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

el cuerpo de la función `suma` define un nuevo ámbito, y cada vez que se llama a `suma` con unos argumentos concretos, la ejecución del programa entra en el cuerpo, lo que crea un nuevo marco que liga sus argumentos con sus parámetros.

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):



El concepto de **entorno** refleja el hecho de que los ámbitos se contienen unos a otros (están anidados unos dentro de otros).

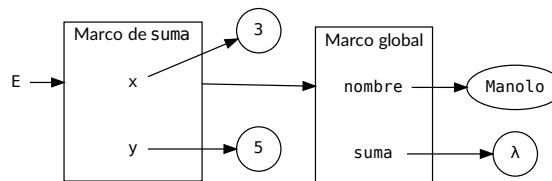
Si un marco A apunta a un marco B, significa que el ámbito de A está contenido en el ámbito de B.

Además:

- El **primer marco** del entorno siempre es el marco del *ámbito más interno* de la instrucción actual.
- El **último marco** siempre es el *marco global*.

En realidad, el marco global apunta, a su vez, a otro marco donde se encuentran las definiciones internas predefinidas del lenguaje (como la función `max`), pero lo ignoraremos de aquí en adelante por simplificar.

Por ejemplo, si en un momento concreto de la ejecución del programa tenemos el siguiente entorno (donde `suma` es una expresión lambda):



Podemos afirmar que:

- El ámbito de la expresión lambda está contenido en el ámbito global.
- El marco actual es el marco de la expresión lambda.
- El ámbito actual es el cuerpo de la expresión lambda.
- Por tanto, el programa se encuentra actualmente ejecutando el cuerpo de la expresión lambda.
- De hecho, está evaluando la llamada `suma(3, 5)`.

2.8. Ligaduras *sombreadas*

¿Qué ocurre cuando una expresión lambda contiene como parámetros nombres que ya están definidos (ligados) en el entorno, en un ámbito más global?

Por ejemplo:

```
1 x = 4
2 total = (lambda x: x * x)(3) # Su valor es 9
```

La `x` que aparece en la línea 1 es distinta a las que aparecen en la 2:

- La `x` de la línea 1 es un identificador ligado a un valor en el ámbito global (el ámbito de creación de esa ligadura es el ámbito global). Esa ligadura, por tanto, se almacena en el marco global, y por eso decimos que esa `x` es *global*.
- Las `x` de la línea 2 son parámetros y variables ligadas de la expresión lambda. Por tanto, el ámbito de esas `x` es *local* al cuerpo de la expresión lambda.

En el ejemplo, el identificador `x` que aparece en el cuerpo de la expresión lambda **está ligado al parámetro `x` de la expresión lambda**.

Por tanto, **no** se refiere al identificador `x` que está fuera de la expresión lambda (y que aquí está ligado al valor `4`), sino al parámetro `x` que, en la llamada de la línea 2, está ligado al valor `3`.

Eso quiere decir que, dentro del cuerpo, `x` vale `3`, no `4`.

Cuando un mismo identificador está ligado en dos ámbitos anidados uno dentro del otro, decimos que:

- El identificador que aparece en el ámbito más externo está **sombreado** (y su ligadura está **sombreada**).
- El identificador que aparece en el ámbito más interno **hace sombra** al identificador sombreado (y su ligadura también se dice que **hace sombra** a la ligadura sombreada).

En nuestro ejemplo, podemos decir que el parámetro `x` de la expresión lambda hace sombra al identificador `x` que aparece en el ámbito global.

Eso significa que no podemos acceder a ese identificador `x` global desde dentro del cuerpo de la expresión lambda como si fuera una variable libre, porque la `x` dentro del cuerpo siempre se referirá a la `x` local (el parámetro de la expresión lambda).

Esto es así porque la primera ligadura del identificador `x` que nos encontramos al recorrer la secuencia de marcos del entorno, buscando un valor para `x`, es la que está en el marco de la expresión lambda, que es el marco actual cuando se está ejecutando su cuerpo.



Entorno en el cuerpo de la expresión lambda, con ligadura sombreada

Si necesitáramos acceder, desde el cuerpo de la expresión lambda, al valor de la `x` que está fuera de la expresión lambda, lo que podemos hacer es **cambiar el nombre** al parámetro `x`. Por ejemplo:

```
x = 4
total = (lambda w: w * x)(3) # Su valor es 12
```

Así, tendremos en la expresión lambda una variable ligada (el parámetro `w`) y una variable libre (el identificador `x` ligado en el ámbito global) al que ahora sí podemos acceder al no estar sombreada y encontrarse dentro del entorno.



Entorno en el cuerpo de la expresión lambda, sin variable sombreada

2.9. Renombrado de parámetros

Los parámetros se pueden *renombrar* (siempre que se haga de forma adecuada) sin que se altere el significado de la expresión lambda.

A esta operación se la denomina **α -conversión**.

Un ejemplo de α -conversión es la que hicimos antes.

La α -conversión hay que hacerla correctamente para evitar efectos indeseados. Por ejemplo, en:

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

si renombramos `x` a `z` tendríamos:

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

lo que es claramente incorrecto. A este fenómeno indeseable se le denomina **captura de variables**.

3. Evaluación

3.1. Evaluación de expresiones con entornos

Al evaluar una expresión, el intérprete **buscará en el entorno el valor al que está ligado cada identificador** que aparezca en la expresión.

Para saber cuánto vale cada identificador, el intérprete buscará **en el primer marco del entorno** una ligadura para ese identificador, y si no la encuentra, **irá subiendo por la secuencia de marcos** hasta encontrarla.

Si no aparece en ningún marco, querrá decir que el identificador no está ligado, o que su ligadura está fuera del entorno, en otro ámbito inaccesible desde el ámbito actual. En cualquier caso, **generará un error** de tipo «*nombre no definido*».

Se debe tener en cuenta, también, las posibles **variables sombreadas** que puedan aparecer.

Si un identificador de un ámbito más local *hace sombra* a otro situado en un ámbito más global, al buscar una ligadura en el entorno se encontrará primero la ligadura más local, ignorando las demás.

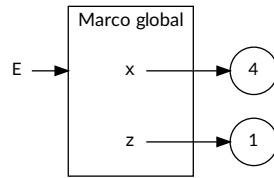
Por ejemplo:

```
1 x = 4
2 z = 1
3 suma = (lambda x, y: x + y + z)(8, 12)
4 y = 3
5 w = 9
```

A medida que vamos ejecutando cada línea del código, tendríamos los siguientes entornos:



Entorno en la línea 1



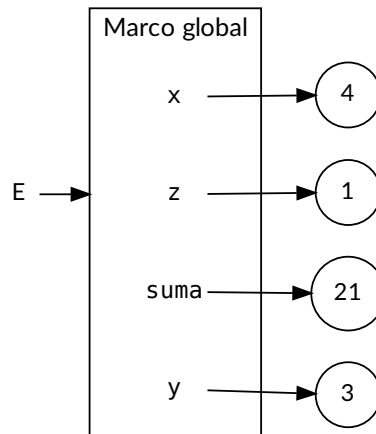
Entorno en la línea 2



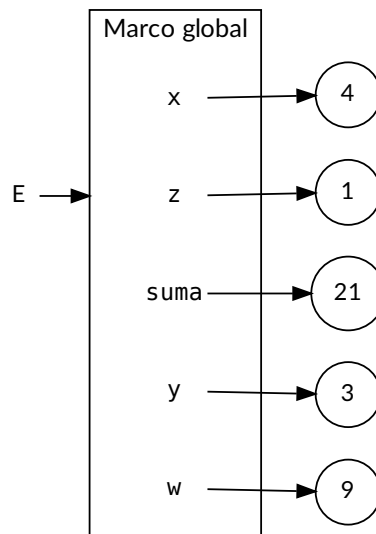
Entorno en la línea 3 en el cuerpo de la expresión lambda, después de aplicar los argumentos y **durante** la ejecución del cuerpo



Entorno en la línea 3 en el cuerpo de la expresión lambda, **después** de ejecutar el cuerpo y devolver el resultado



Entorno en la línea 4



Entorno en la línea 5

3.2. Evaluación de expresiones lambda con entornos

Para que una expresión lambda funcione, sus variables libres deben estar ligadas a algún valor en el entorno **en el momento de evaluar la aplicación de la expresión lambda sobre unos argumentos**.

Por ejemplo:

```

1 >>> prueba = lambda x, y: x + y + z # aquí no da error
2 >>> prueba(4, 3) # aquí sí
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   File "<stdin>", line 1, in <lambda>
6 NameError: name 'z' is not defined

```

da error porque `z` no está definido (no está ligado a ningún valor en el entorno) en el momento de llamar a `prueba` en la línea 2.

En cambio:

```

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

```

sí funciona (y devuelve `16`) porque, en el momento de evaluar la aplicación de la expresión lambda (en la línea 3), el identificador `z` está ligado a un valor en el entorno (en este caso, `9`).

Observar que no es necesario que las variables libres estén ligadas en el entorno cuando *se crea* la expresión lambda, sino cuando *se evalúa el cuerpo de la expresión lambda*, o sea, cuando se aplica la expresión lambda a unos argumentos.

Ejemplo

En el siguiente *script*:

```

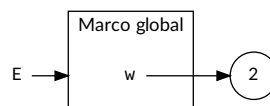
1 w = 2
2 f = lambda x, y: 5 + (lambda z: z + 3)(x + y)
3 r = f(2, 4)
4 m = (lambda x: x ** 2)(3)

```

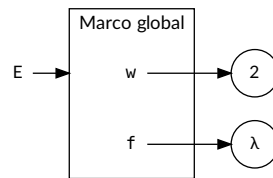
existen cuatro ámbitos:



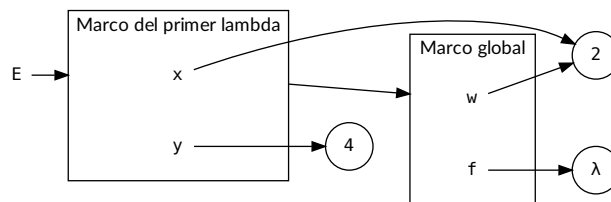
Su ejecución, línea a línea, produce los siguientes entornos:



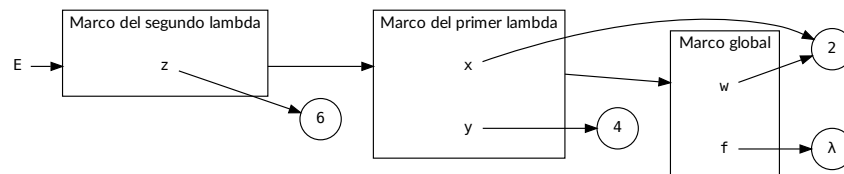
Entorno en la línea 1



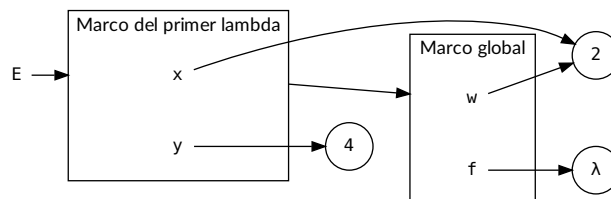
Entorno en la línea 2



Entorno en la línea 3 en el cuerpo de la primera expresión lambda, después de aplicar sus argumentos y durante la ejecución de su cuerpo



Entorno en la línea 3 en el cuerpo de la segunda expresión lambda, después de aplicar sus argumentos y durante la ejecución de su cuerpo



Entorno en la línea 3 en el cuerpo de la segunda expresión lambda, después de ejecutar su cuerpo y devolver su resultado



Entorno en la línea 3 en el cuerpo de la primera expresión lambda, después de ejecutar su cuerpo y devolver su resultado



Entorno en la línea 4 en el cuerpo de la tercera expresión lambda, después de aplicar sus argumentos y durante la ejecución de su cuerpo



Entorno en la línea 4 en el cuerpo de la tercera expresión lambda, después de ejecutar su cuerpo y devolver su resultado

3.2.1. Visualización en Pythontutor

Pythontutor es una herramienta online muy interesante y práctica que nos permite ejecutar un *script* paso a paso y visualizar sus efectos.

Muestra la pila de control, los marcos dentro de ésta, las ligaduras dentro de éstos y los datos almacenados en el montículo.

Entrando en <http://pythontutor.com/visualize.html> se abre un área de texto donde se puede teclear (o copiar y pegar) el código fuente del *script* a ejecutar.

Pulsando en «*Visualize Execution*» se pone en marcha, pudiendo ejecutar todo el *script* de una vez o hacerlo paso a paso.

Conviene elegir las siguientes opciones:

- *Hide exited frames (default)*
- *Render all objects on the heap (Python/Java)*
- *Draw pointers as arrows (default)*

Visualizar el *script* anterior en Pythontutor

Ejercicio

1. En el *script* anterior:

```

1 w = 2
2 f = lambda x, y: 5 + (lambda z: z + 3)(x + y)
3 r = f(2, 4)
4 m = (lambda x: x ** 2)(3)
  
```

indicar:

- a. Los identificadores.
- b. Los ámbitos.
- c. Los entornos, marcos y ligaduras en cada línea de código.
- d. Los ámbitos de cada ligadura.
- e. Los ámbitos de creación de cada ligadura.
- f. Los ámbitos de cada aparición de cada identificador.
- g. Las ligaduras sombreadas y los identificadores sombreados.
- h. Los identificadores y ligaduras que hacen sombra.

3.3. Estrategias de evaluación

A la hora de evaluar una expresión (cualquier expresión) existen varias **estrategias** diferentes que se pueden adoptar.

Cada lenguaje implementa sus propias estrategias de evaluación que están basadas en las que vamos a ver aquí.

Básicamente se trata de decidir, en cada paso de reducción, qué subexpresión hay que reducir, en función de:

- El orden de evaluación:
 - * De fuera adentro o de dentro afuera.
 - * De izquierda a derecha o de derecha a izquierda.
- La necesidad o no de evaluar dicha subexpresión.

3.3.1. Orden de evaluación

En un lenguaje de programación funcional puro se cumple la **transparencia referencial**, según la cual el valor de una expresión depende sólo del valor de sus subexpresiones (también llamadas *redexes*, del inglés, *reducible expression*).

Pero eso también implica que **no importa el orden en el que se evalúen las subexpresiones**: el resultado debe ser siempre el mismo.

Gracias a ello podemos usar nuestro modelo de sustitución como modelo computacional.

Hay dos **estrategias básicas de evaluación**:

- **Orden aplicativo**: reducir siempre el *redex* más **interno** (y más a la izquierda).
- **Orden normal**: reducir siempre el *redex* más **externo** (y más a la izquierda).

Python usa el orden aplicativo, salvo excepciones.

3.3.1.1. Orden aplicativo

El **orden aplicativo** consiste en evaluar las expresiones *de dentro afuera*, es decir, empezando por el *redex* más **interno** y a la izquierda.

El *redex* más interno es el que no contiene a otros *redexes*. Si existe más de uno que cumpla esa condición, se elige el que está más a la izquierda.

Eso implica que los operandos y los argumentos se evalúan **antes** que los operadores y las aplicaciones de funciones.

Corresponde a lo que en muchos lenguajes de programación se denomina **paso de argumentos por valor** (*call-by-value*).

Por ejemplo, si tenemos la siguiente función:

```
cuadrado = lambda x: x * x
```

según el orden aplicativo, la expresión `cuadrado(3 + 4)` se reduce así:

```
cuadrado(3 + 4)           # definición de cuadrado
= (lambda x: x * x)(3 + 4) # evalúa 3 y devuelve 3
= (lambda x: x * x)(3 + 4) # evalúa 4 y devuelve 4
= (lambda x: x * x)(3 + 4) # evalúa 3 + 4 y devuelve 7
= (lambda x: x * x)(7)     # aplicación a 7
= (7 * 7)                  # evalúa (7 * 7) y devuelve 49
= 49
```

3.3.1.2. Orden normal

El **orden normal** consiste en evaluar las expresiones *de fuera adentro*, es decir, empezando siempre por el *redex* más **externo** y a la izquierda.

El *redex* más externo es el que no está contenido en otros *redexes*. Si existe más de uno que cumpla esa condición, se elige el que está más a la izquierda.

Eso implica que los operandos y los argumentos se evalúan **después** de las aplicaciones de los operadores y las funciones.

Por tanto, los argumentos que se pasan a las funciones lo hacen **sin evaluarse** previamente.

Corresponde a lo que en muchos lenguajes de programación se denomina **paso de argumentos por nombre** (*call-by-name*).

Por ejemplo, si tenemos la siguiente función:

```
cuadrado = lambda x: x * x
```

según el orden normal, la expresión `cuadrado(3 + 4)` se reduce así:

```
cuadrado(3 + 4)           # definición de cuadrado
= (lambda x: x * x)(3 + 4) # aplicación a (3 + 4)
= ((3 + 4) * (3 + 4))      # evalúa 3 y devuelve 3
```

```

= ((3 + 4) * (3 + 4))      # evalúa 4 y devuelve 4
= ((3 + 4) * (3 + 4))      # evalúa (3 + 4) y devuelve 7
= 7 * (3 + 4)              # evalúa 3 y devuelve 3
= 7 * (3 + 4)              # evalúa 4 y devuelve 4
= 7 * (3 + 4)              # evalúa (3 + 4) y devuelve 7
= 7 * 7                    # evalúa 7 * 7 y devuelve 49
= 49

```

3.3.2. Composición de funciones

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

```

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

```

La expresión `area(11 + 1)` se evaluaría así según el *orden aplicativo*:

```

1 area(11 + 1)                # definición de area
2 = (lambda r: 3.1416 * cuadrado(r))(11 + 1) # evalúa 11 y devuelve 11
3 = (lambda r: 3.1416 * cuadrado(r))(11 + 1) # evalúa 1 y devuelve 1
4 = (lambda r: 3.1416 * cuadrado(r))(11 + 1) # evalúa 11 + 1 y devuelve 12
5 = (lambda r: 3.1416 * cuadrado(r))(12)      # aplicación a 12
6 = (3.1416 * cuadrado(12))                  # evalúa 3.1416 y devuelve 3.1416
7 = (3.1416 * cuadrado(12))                  # definición de cuadrado
8 = (3.1416 * (lambda x: x * x)(12))          # aplicación a 12
9 = (3.1416 * (12 * 12))                     # evalúa (12 * 12) y devuelve 144
10 = (3.1416 * 144)                          # evalúa (3.1416 * 11) y...
11 = 452.3904                                # ... devuelve 452.3904

```

En detalle:

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

La expresión `area(11 + 1)` se evaluaría así según el *orden normal*:

```

1 area(11 + 1)                # definición de area
2 = (lambda r: 3.1416 * cuadrado(r))(11 + 1) # aplicación a (11 + 1)
3 = (3.1416 * cuadrado(11 + 1))              # evalúa 3.1416 y devuelve 3.1416
4 = (3.1416 * cuadrado(11 + 1))              # definición de cuadrado
5 = (3.1416 * (lambda x: x * x)(11 + 1))     # aplicación a (11 + 1)

```



```

6 = (3.1416 * ((11 + 1) * (11 + 1)))      # evalúa (11 + 1) y devuelve 12
7 = (3.1416 * (12 * (11 + 1)))           # evalúa (11 + 1) y devuelve 12
8 = (3.1416 * (12 * 12))                 # evalúa (12 * 12) y devuelve 144
9 = (3.1416 * 144)                       # evalúa (3.1416 * 144) y...
10 = 452.3904                            # ... devuelve 452.3904

```

En ambos casos (orden aplicativo y orden normal) se obtiene el mismo resultado.

En detalle:

- **Línea 1:** Se evalúa el *redex* más externo, que es `area(11 + 1)`. Para ello, se reescribe la definición de `area...`
- **Línea 2:** ... y se aplica la expresión lambda al argumento `11 + 1`.
- **Línea 3:** El *redex* más externo es el `*`, pero para evaluarlo hay que evaluar primero todos sus argumentos, por lo que primero se evalúa el izquierdo, que es `3.1416`.
- **Línea 4:** Ahora hay que evaluar el derecho (`cuadrado(11 + 1)`), por lo que se reescribe la definición de `cuadrado...`
- **Línea 5:** ... y se aplica la expresión lambda al argumento `11 + 1`.
- Lo que queda es todo aritmética.

A veces no resulta fácil determinar si un *redex* es más interno o externo que otro, sobre todo cuando se mezclan funciones y operadores en una misma expresión.

En ese caso, puede resultar útil reescribir los operadores como funciones, cuando sea posible.

Por ejemplo, la siguiente expresión:

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

se puede reescribir como:

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

lo que muestra claramente que la suma es más externa que el valor absoluto y el máximo (que están, a su vez, al mismo nivel de profundidad).

Un ejemplo más complicado:

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

se reescribiría como:

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

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

1. Suma (+ o `add`).

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

3.3.3. Evaluación estricta y no estricta

Existe otra forma de ver la evaluación de una expresión:

- **Evaluación estricta o *impaciente***: Reducir todos los *redexes* aunque no hagan falta para calcular el valor de la expresión.
- **Evaluación no estricta o *perezosa***: Reducir sólo los *redexes* que sean estrictamente necesarios para calcular el valor de la expresión.

Ejemplo

Sabemos que la expresión `1 / 0` da un error de *división por cero*:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Supongamos que tenemos la siguiente definición:

```
primero = lambda x, y: x
```

de forma que `primero` es una función que simplemente devuelve el primero de sus argumentos.

Es evidente que la función `primero` no necesita evaluar nunca su segundo argumento, ya que no lo utiliza (simplemente devuelve el primero de ellos). Por ejemplo, `primero(4, 3)` devuelve `4`.

Sabiendo eso... ¿qué valor devolvería la siguiente expresión?

```
primero(4, 1 / 0)
```

Curiosamente, el resultado dependerá de si la evaluación es estricta o perezosa:

- **Si es estricta**, el intérprete evaluará todos los argumentos de la expresión lambda aunque no se utilicen luego en su cuerpo. Por tanto, al evaluar `1 / 0` devolverá un error.

Es lo que ocurre cuando se evalúa siguiendo el **orden aplicativo**.

- En cambio, **si es perezosa**, el intérprete evaluará únicamente aquellos argumentos que se usen en el cuerpo de la expresión lambda, y en este caso sólo se usa el primero, así que dejará sin evaluar el segundo, no dará error y devolverá directamente `4`.

Es lo que ocurre cuando se evalúa siguiendo el **orden normal**:

```
primero(4, 1 / 0) = (lambda x, y: x)(4, 1 / 0) = (4) = 4
```

Hay un resultado teórico que avala lo que acabamos de observar:

Teorema de estandarización:

Si una expresión tiene forma normal, el **orden normal** de evaluación conduce seguro a la misma.

En cambio, el orden aplicativo es posible que no encuentre la forma normal de la expresión.

En **Python** la evaluación es **estricta**, salvo algunas excepciones:

- El operador ternario:

```
<expr_condicional> ::= <valor_si_cierto> if <condición> else <valor_si_falso>
```

evalúa perezosamente *<valor_si_cierto>* y *<valor_si_falso>* dependiendo del valor de la *<condición>*.

- Los operadores lógicos **and** y **or** también son perezosos (se dice que evalúan **en cortocircuito**):

- * **True or** *x*

- siempre es igual a **True**.

- * **False and** *x*

- siempre es igual a **False**.

En ambos casos no es necesario evaluar *x*.

En Java también existe un operador ternario (**? :**) y unos operadores lógicos (**||** y **&&**) que se evalúan de igual forma que en Python.

La mayoría de los lenguajes de programación usan evaluación estricta y paso de argumentos por valor (siguen el orden aplicativo).

Haskell, por ejemplo, es un lenguaje funcional puro que usa evaluación perezosa y sigue el orden normal.

La evaluación perezosa en Haskell permite resultados muy interesantes, como la posibilidad de manipular estructuras de datos infinitas.

4. Abstracciones funcionales

4.1. Pureza

Si el cuerpo de una expresión lambda no contiene variables 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 sí contiene variables libres, el valor que obtendremos al aplicarla a unos argumentos no sólo dependerá del valor de esos argumentos, sino también de los valores a los que estén ligadas las variables 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 una variable 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*:

```
>>> 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 *suma3*, por lo que es una variable libre en el cuerpo de la expresión lambda de *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, es decir, de sus parámetros o variables ligadas.

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

- Una expresión lambda que contiene **sólo variables ligadas** es **más pura** que otra que también contiene variables libres.
- Una expresión lambda cuyas **variables libres** representan **funciones** que se usan en el cuerpo de la expresión lambda, es **más pura** que otra cuyas variables 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 única variable libre (*suma*) se usa como una **función**, y las funciones tienden a no cambiar durante la ejecución del programa, al contrario que los demás tipos de valores.

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 variables libres (en este
# caso, sólo una: producto) son funciones:
cuadrado = lambda x: producto(x, x)
# suma es impura, porque su variable 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.

4.2. 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).

Podemos escribir `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`, 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.

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 la podemos arreglar sin definir el 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 elevar 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 (*elevar al cubo*).

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

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**.

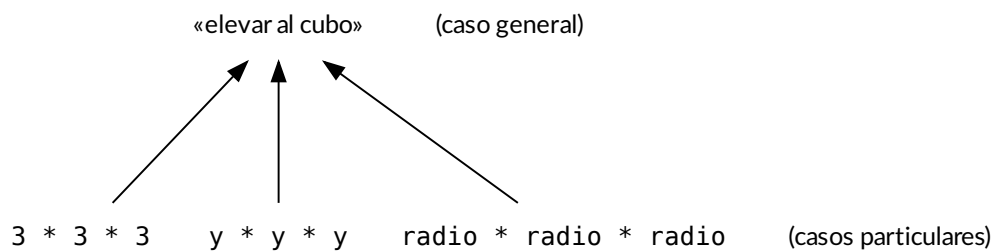
Crear casos generales a partir de patrones que se repiten en casos particulares es una de las principales razones por las que creamos abstracciones.

Otras veces 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.

Y otras veces simplemente cuando queremos que nuestro programa pueda **expresar un concepto abstracto**, como el de «elevant 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 «elevant 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.

Un programador no debe necesitar saber cómo está implementada una función para poder usarla.

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 haya 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.

4.2.1. 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 está formada por tres propiedades fundamentales:

- El **dominio**: el conjunto de argumentos válidos.
- El **rango**: 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 hemos llamado «**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*.

Pero para especificar una función, en cambio, 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 llamar a la función.

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 `n2`.

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).

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.

Otro ejemplo más completo:

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

`count` es una **función oculta o auxiliar** (en este caso, un *método auxiliar*). Las funciones auxiliares se puede usar en la especificación siempre que estén perfectamente especificadas, aunque no estén implementadas.

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

Además, estamos diciendo que devuelve el mismo resultado que devuelve el método `count`.

Es decir: cuenta el número de veces que el carácter `car` aparece en `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

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.

5. Computabilidad

5.1. Funciones y procesos

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

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

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

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

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

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

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

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

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

5.2. Funciones *ad-hoc*

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

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

ABC, ACB, BAC, BCA, CAB, CBA

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

La implementación de esa función deberá satisfacer la siguiente especificación:

$$\left\{ \begin{array}{l} \text{Pre : } n \geq 0 \\ \quad \text{permutas}(n : \text{int}) \rightarrow \text{int} \\ \text{Post : } \text{permutas}(n) = \text{el número de permutaciones que} \\ \quad \text{podemos formar con } n \text{ elementos} \end{array} \right.$$

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

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

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

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

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

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

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

- Realmente son **tramposos** (no calculan nada).
- **No son útiles**, porque al final el cálculo se tiene que hacer con otra cosa.
- Generalmente resulta **imposible** que una función de este tipo abarque todos los posibles datos de entrada, ya que, en principio, puede haber **infinitos** y, por tanto, su código fuente también tendría que ser infinito.

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

5.3. Funciones recursivas

5.3.1. 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

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

5.3.2. 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**.

5.3.3. 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.

5.3.4. Diseño de funciones recursivas

El diseño de funciones recursivas se basa en:

- Pensamiento optimista
- Descomposición (reducción) del problema
- Identificación de problemas no reducibles (mínimos)

5.3.4.1. Pensamiento optimista

Consiste en suponer que la función deseada ya existe y es capaz de resolver ejemplares más pequeños del problema (este paso se denomina **hipótesis inductiva**).

Se trata de encontrar el patrón común de forma que resolver el problema principal implique el mismo patrón en un problema más pequeño.

Ejemplo:

- Queremos diseñar una función que calcule el factorial de un número.
- Para ello, supongamos que ya contamos con una función que calcula el factorial de un número más pequeño. Tenemos que creer y confiar en que es así, aunque ahora mismo no sea verdad.

Es decir: si queremos calcular el factorial de n , suponemos que tenemos ya una función *fact* que no sabe calcular el factorial de n , pero sí el de $(n - 1)$. *Ésta es nuestra hipótesis inductiva.*

5.3.4.2. Descomposición del problema

Reducimos el problema de forma que así tendremos un ejemplar más pequeño del mismo problema y, por tanto, podremos usar la función *fact* anterior para poder resolver ese ejemplar más pequeño.

A continuación, usamos dicha solución *parcial* para obtener la solución al problema original.

Ejemplo:

- Sabemos que $n! = n \cdot (n - 1)!$
- Sabemos que la función *fact* sabe calcular el factorial de $(n - 1)$ (por pensamiento optimista).
- Por tanto, lo único que tenemos que hacer para obtener el factorial de n es multiplicar n por el resultado de *fact*($n - 1$).

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

5.3.4.3. Identificación de problemas no reducibles

Debemos identificar los ejemplares más pequeños (los que no se pueden reducir más) para los cuales hay una solución explícita y directa que no necesita recursividad: los *casos base*.

Es importante comprobar que la reducción que le hemos realizado al problema en el paso anterior produce ejemplares que están más cerca del caso base.

Ejemplo:

- En nuestro caso, sabemos que $0! = 1$, por lo que nuestra función podría devolver directamente 1 cuando se le pida calcular el factorial de 0.
- Además, en la reducción obtenida en el paso anterior, pasamos de calcular el factorial de n a calcular el factorial de uno menos, con lo cual, cada vez estaremos más cerca del caso base, que es el factorial de 0. Al final siempre acabaremos alcanzando el caso base.

Combinando todos los pasos, obtenemos la solución general:

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

5.3.5. 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*:

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

5.3.5.1. 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**.

5.3.5.2. 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 n	Proporcional a n
Iterativo	Proporcional a n	Constante

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

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

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

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

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

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

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

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

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

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

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

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

En este caso, las dos implementaciones son:

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

y

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

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

5.3.6. 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:

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

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

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

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:

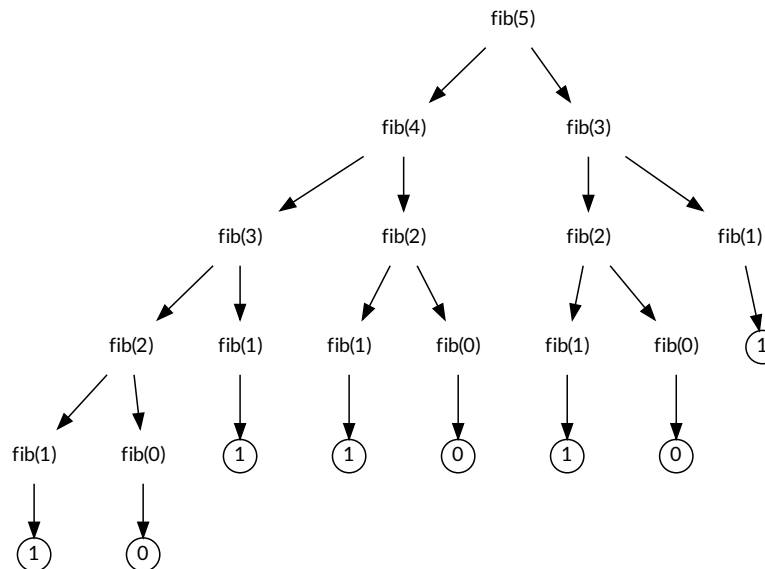
```
fib = lambda n: 0 if n == 0 else \
    1 if n == 1 else \
    fib(n - 1) + fib(n - 2)
```

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 `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 `fib(n)` y `fib(n + 1)`, respectivamente.

En Python sería:

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

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

5.3.7. 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:

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

5.4. La pila de control

La **pila de control** es una estructura de datos que utiliza el intérprete para llevar la cuenta de las **llamadas activas** en un determinado momento, incluyendo el valor de sus parámetros y el punto de retorno al que debe devolverse el control cuando finalice la ejecución de la función.

- Las **llamadas activas** son aquellas llamadas a funciones que aún no han terminado de ejecutarse.

La pila de control es, básicamente, un **almacén de marcos**.

Cada vez que se hace una nueva llamada a una función, **su marco** correspondiente **se almacena en la cima de la pila** sobre los demás marcos que pudiera haber.

Ese marco es el primero de la secuencia de marcos que forman el entorno de la función, que estarán almacenados más abajo en la pila.

Los marcos se enlazan entre sí para representar los entornos que actúan en las distintas llamadas activas.

El intérprete puede, además, almacenar ahí cualquier otra información que necesite para gestionar las llamadas a funciones.

El marco de la función, junto con toda esa información adicional, se denomina **registro de activación**.

Por tanto, **la pila de control almacena registros de activación**.

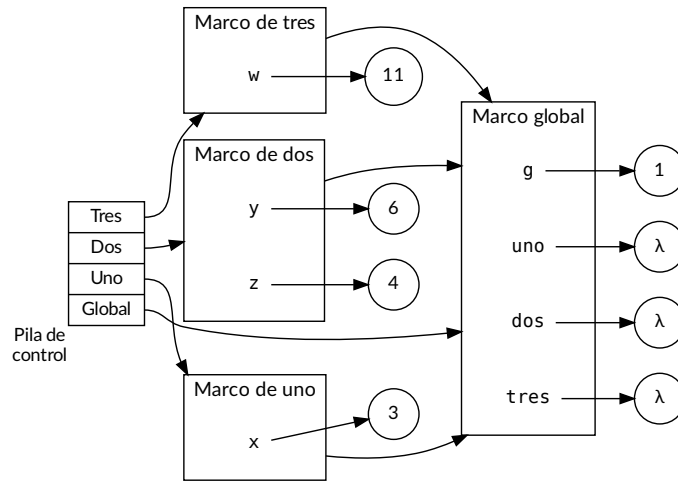
Cada llamada activa está representada por su correspondiente registro de activación en la pila, y cada registro de activación va asociado a un marco.

En cuanto la llamada finaliza, su registro de activación se saca de la pila y se transfiere el control a la llamada que está inmediatamente debajo (si es que hay alguna).

Cuando desaparece un registro de activación, también se elimina de la memoria su marco asociado (hay una excepción a esto, que veremos en posteriores temas cuando hablemos de las *clausuras*).

Ejemplos

```
g = 1
uno = lambda x: 1 + dos(2 * x, 4)
dos = lambda y, z: tres(y + z + g)
tres = lambda w: "W vale " + str(w)
uno(3)
```

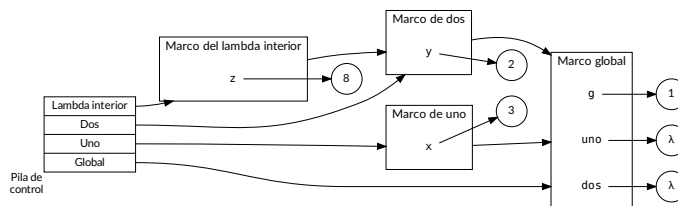
Pila de control con la llamada a la función `tres` activada

Del análisis del diagrama del ejemplo anterior se pueden deducir las siguientes conclusiones:

- En un momento dado, dentro del ámbito global se ha llamado a la función `uno`, la cual ha llamado a la función `dos`, la cual ha llamado a la función `tres`, la cual aún no ha terminado de ejecutarse.
- El entorno en la función `uno` empieza por el marco de `uno`, el cual apunta al marco global.
- El entorno en la función `dos` empieza por el marco de `dos`, el cual apunta al marco global.
- El entorno en la función `tres` empieza por el marco de `tres`, el cual apunta al marco global.

Si tenemos ámbitos anidados, los marcos se apuntarán entre sí en el entorno. Por ejemplo:

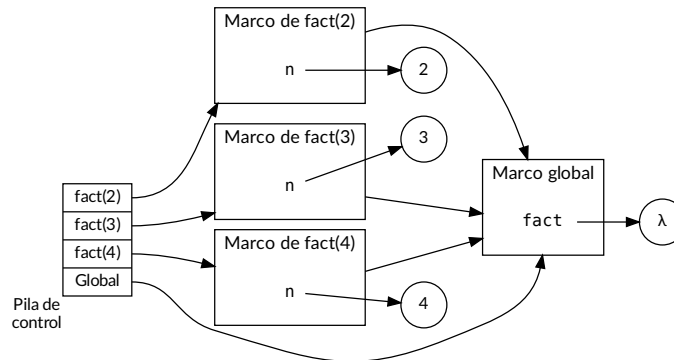
```
g = 1
uno = lambda x: dos(x - 1)
dos = lambda y: 1 + (lambda z: z * 2)(y ** 3)
uno(3)
```

Pila de control con ámbitos anidados y la función `dos` activada

Hemos dicho que habrá un registro de activación por cada nueva llamada que se realice a una función, y que ese registro se mantendrá en la pila hasta que la llamada finalice.

Por tanto, en el caso de una función recursiva, tendremos un registro de activación por cada llamada recursiva.

```
fact = lambda n: 1 if n == 0 else n * fact(n - 1)
fact(4)
```



Pila de control tras tres activaciones desde `fact(4)`

Los **traductores que optimizan la recursividad final** lo que hacen es sustituir cada llamada recursiva por la siguiente llamada recursiva a la misma función.

De esta forma, el marco que genera cada nueva llamada recursiva no se apila sobre los marcos anteriores en la pila, sino que sustituye al marco de la llamada que la ha llamado a ella.

Por ejemplo, en el siguiente caso:

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

fact(5)
= fact_iter(5, 1)
= fact_iter(4, 5)
= fact_iter(3, 20)
= fact_iter(2, 60)
= fact_iter(1, 120)
= fact_iter(0, 120)
= 120
```

`fact_iter(4, 5)` llama a `fact_iter(3, 20)` y devuelve directamente el resultado de ésta.

Es decir: `fact_iter(4, 5) == fact_iter(3, 20)`, así que hacer `fact_iter(4, 5)` es lo mismo que hacer `fact_iter(3, 20)`.

Por tanto, la llamada a `fact_iter(4, 5)` se puede sustituir por la llamada a `fact_iter(3, 20)`.

Un intérprete que optimiza la recursividad final no apilaría el marco de la segunda llamada sobre el marco de la primera, sino que el marco de la segunda sustituiría al marco de la primera dentro de la pila.

Así se haría también con las demás llamadas recursivas a `fact_iter(2, 60)`, `fact_iter(1, 120)` y `fact_iter(0, 120)`.

De este modo, la pila no crecería con cada nueva llamada recursiva.

5.5. Un lenguaje Turing-completo

El paradigma funcional que hemos visto hasta ahora (uno que nos permite definir funciones, componer dichas funciones y aplicar recursividad, junto con el operador ternario condicional) es un lenguaje de programación **completo**.

Decimos que es **Turing completo**, lo que significa que puede computar cualquier función que pueda computar una máquina de Turing.

Como las máquinas de Turing son los ordenadores más potentes que podemos construir (ya que describen lo que cualquier ordenador es capaz de hacer), esto significa que nuestro lenguaje puede calcular todo lo que pueda calcular cualquier ordenador.

6. Tipos de datos recursivos

6.1. Cadenas

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

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

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

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

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

6.2. Tuplas

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

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

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

Por ejemplo:

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

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

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

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

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

Según el ejemplo anterior:

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

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

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

Por ejemplo:

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

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

6.3. Rangos

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

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

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

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

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

El valor de `stop` no se alcanza nunca.

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

`step` debe ser siempre distinto de cero.

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

Ejemplos

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

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

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

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

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

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

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

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

```
>>> range(2, 5, 1)
range(2, 5)
>>> range(2, 5, 2)
range(2, 5, 2)
```

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

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

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

Los rangos también pueden verse como un **tipo de datos recursivo**, ya que todo rango `r`:

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

Según el ejemplo anterior:

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

6.4. Conversión a tupla

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

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

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

7. Funciones de orden superior

7.1. Concepto

Sabemos que, en programación funcional, las funciones son un dato más, como cualquier otro (es decir: tienen un tipo, se pueden ligar a identificadores, etcétera).

Pero eso significa que también se pueden pasar como argumentos a otras funciones o se pueden devolver como resultado de otras funciones.

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

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

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

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

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

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

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

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

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

Tanto `aplica5` como `suma_o Resta` son **funciones de orden superior**.

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

El mismo concepto se puede aplicar a casos particulares de funciones, y al hacerlo damos un paso

más en nuestro camino hacia un mayor grado de abstracción.

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

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

Supongamos las dos funciones siguientes:

```
# Suma los enteros comprendidos entre a y b:
suma_enteros = lambda a, b: 0 if a > b else a + suma_enteros(a + 1, b)

# Suma los cubos de los enteros comprendidos entre a y b:
suma_cubos = lambda a, b: 0 if a > b else cubo(a) + suma_cubos(a + 1, b)
```

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

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

Podríamos haber escrito las funciones anteriores rellenando los «casilleros» del siguiente *patrón general*:

```
<nombre> = lambda a, b: 0 if a > b else <término>(a) + <nombre>(a + 1, b)
```

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

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

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

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

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

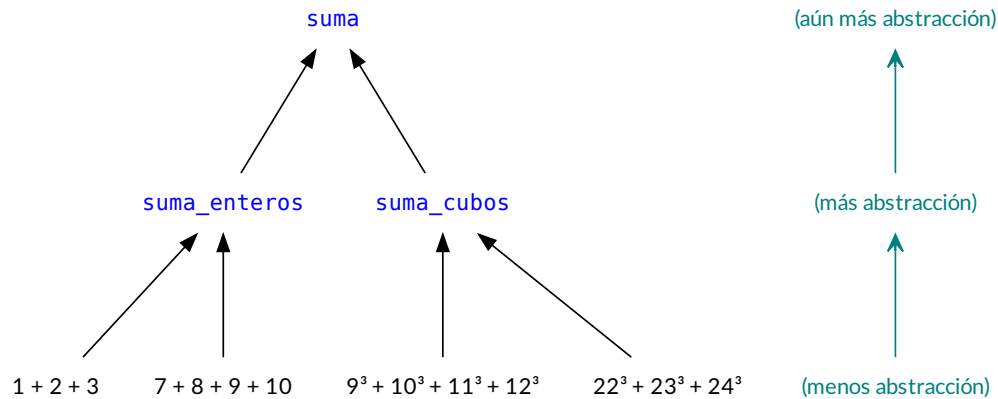
En programación funcional lo conseguimos creando funciones que conviertan los «casilleros» en parámetros que recibirían funciones:

```
suma = lambda term, a, b: 0 if a > b else term(a) + suma(term, a + 1, b)
```

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

```
suma_enteros = lambda a, b: suma(lambda x: x, a, b)
suma_cubos = lambda a, b: suma(lambda x: x * x * x, a, b)
# O mejor aún:
suma_cubos = lambda a, b: suma(cubo, a, b)
```

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



El camino de subida hacia una abstracción cada vez mayor

Ejercicio

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

7.2. [map](#)

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

Ejercicio

5. Inténtalo.

Una forma de hacerlo sería:

```
eleva_cubo = lambda t: () if t == () else \
    (cubo(t[0]),) + eleva_cubo(t[1:])
```

¿Y elevar a la cuarta potencia?

```
elevar_cuarta = lambda t: () if t == () else \
    ((lambda x: x ** 4)(t[0]),) + elevar_cuarta(t[1:])
```

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

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

```
map(func, iterable)
```

donde:

- `func` debe ser una función de un solo argumento.
- `iterable` puede ser cualquier cosa compuesta de elementos que se puedan recorrer de uno en uno, como una **tupla**, una **cadena** o un **rango** (cualquier *secuencia* de elementos nos vale).

Podemos usarla así:

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

Lo que devuelve no es una tupla, sino un objeto **iterador** que examinaremos con más detalle más adelante.

Por ahora, nos basta con saber que un iterador es un flujo de datos que se pueden recorrer de uno en uno.

Lo que haremos aquí será transformar ese iterador en la tupla correspondiente usando la función `tuple` sobre el resultado de `map`:

```
>>> tuple(map(cubo, (1, 2, 3, 4)))
(1, 8, 27, 64)
```

Además de una tupla, también podemos usar un rango como argumento para `map`:

```
>>> tuple(map(cubo, range(1, 5)))
(1, 8, 27, 64)
```

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

Ejercicio

6. Inténtalo.

Podríamos definirla así:

```
map = lambda f, t: () if t == () else (f(t[0]),) + map(f, t[1:])
```

7.3. `filter`

`filter` es una **función de orden superior** que devuelve aquellos elementos de una tupla (o cualquier cosa *iterable*) que cumplen una determinada condición.

Su signature es:

```
filter(function, iterable)
```

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

Como `map`, también devuelve un *iterador*, que se puede convertir a tupla con la función `tuple`.

Por ejemplo:

```
>>> tuple(filter(lambda x: x > 0, (-4, 3, 5, -2, 8, -3, 9)))
(3, 5, 8, 9)
```

7.4. `reduce`

`reduce` es una **función de orden superior** que aplica, de forma acumulativa, una función a todos los elementos de una tupla (o cualquier cosa *iterable*).

Las operaciones se hacen agrupándose **por la izquierda**.

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

Por ejemplo, para calcular la suma de todos los elementos de una tupla, haríamos:

```
>>> suma = lambda t: 0 if t == () else t[0] + suma(t[1:])
>>> suma((1, 2, 3, 4))
10
```

Y para calcular el producto:

```
>>> producto = lambda t: 1 if t == () else t[0] * producto(t[1:])
>>> producto((1, 2, 3, 4))
24
```

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

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

Eso es lo que hace la función `reduce`.

Su signature es:

```
reduce(function, sequence [, initial])
```

donde:

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

Para usarla, tenemos que *importarla* previamente del módulo `functools`.

- No es la primera vez que importamos un módulo. Ya lo hicimos con el módulo `math`.
- En su momento estudiaremos con detalle qué son los módulos. Por ahora nos basta con lo que ya sabemos: que contienen definiciones que podemos incorporar a nuestros *scripts*.

Por ejemplo, para calcular la suma y el producto de `(1, 2, 3, 4)`:

```
from functools import reduce
tupla = (1, 2, 3, 4)
suma_de_numeros = reduce(lambda x, y: x + y, tupla, 0)
producto_de_numeros = reduce(lambda x, y: x * y, tupla, 1)
```

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

Ejercicio

7. Intentalo.

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

```
reduce = lambda fun, tupla, ini: ini if tupla == () else \
    tupla[0] if tupla[1:] == () else \
    fun(tupla[0], reduce(fun, tupla[1:], ini))
```

7.5. Expresiones generadoras

Dos operaciones que se realizan con frecuencia sobre una estructura iterable son:

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

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

El resultado que devuelve es un iterador que (como ya sabemos) podemos convertir fácilmente en una tupla usando la función `tuple`.

Por ejemplo:

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

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

Su sintaxis es:

```
<expr_gen> ::= (<expresión> for <identificador> in <secuencia> [if <condición>])+
```

Los elementos de la salida generada serán los sucesivos valores de *<expresión>*.

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

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

Por ejemplo:

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

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.