Programación funcional (II)

Ricardo Pérez López

IES Doñana, curso 2021/2022

Generado el 2022/07/17 a las 14:10:00

Índice

1. Computabilidad

1.1. 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.

1.2. 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:

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

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

```
permutas = lambda n: 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.

1.3. Funciones recursivas

1.3.1. Definición

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

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

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

Ese es el tipo de recursividad que vamos a estudiar.

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

Por ejemplo:

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

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

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

Por ejemplo:

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

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

Ejercicio

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

1.3.2. 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**.

1.3.3. 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:

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

Por ejemplo:

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

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

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

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

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

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

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

$$0! = 1$$

Combinando ambos casos tendríamos:

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

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

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

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

```
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.

1.3.4. Diseño de funciones recursivas

El diseño de funciones recursivas se basa en:

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

1.3.4.1. 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.*

1.3.4.2. 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.**

1.3.4.3. 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:

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

1.3.5. 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*:

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

1.3.5.1. 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***.

1.3.5.2. 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

En este caso, las dos implementaciones son:

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

y

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

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

1.3.6. 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:

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

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

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

Y su implementación en Python podría ser:

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

o bien, separando la definición en varias líneas:

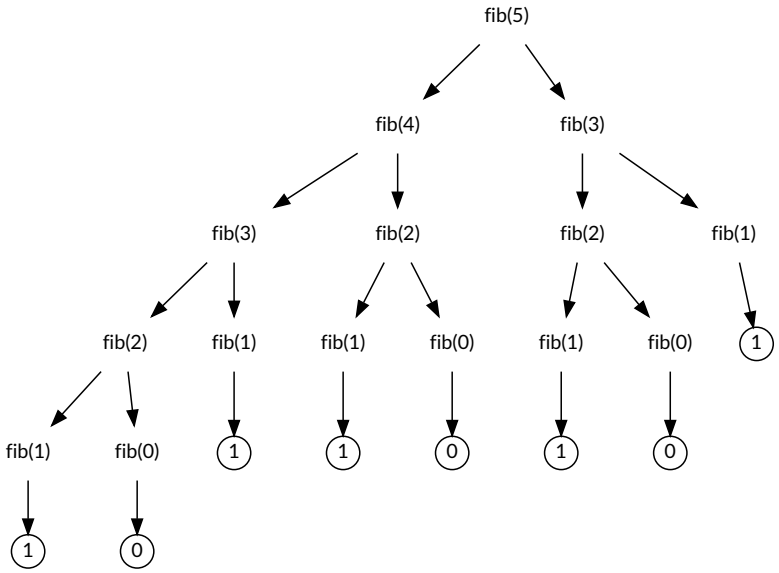
```
fib = lambda n: 0 if n == 0 else \
                1 if n == 1 else \
                fib(n - 1) + fib(n - 2)
```

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

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

En general, el proceso resultante tiene forma de árbol.

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



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

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

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

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

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

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

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

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

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

En Python sería:

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

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

1.3.7. 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.

1.4. 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.

- 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 también estarán almacenados en la pila, más abajo.

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

El intérprete almacena en el marco cualquier información que necesite para gestionar las llamadas a funciones, incluyendo:

- Las ligaduras entre los parámetros y sus valores (por supuesto).
- La ligadura que apunta al valor de retorno de la función.
- Cuál es el siguiente marco que le sigue en el entorno.
- El punto de retorno, dentro del programa, al que debe devolverse el control cuando finalice la ejecución de la función.

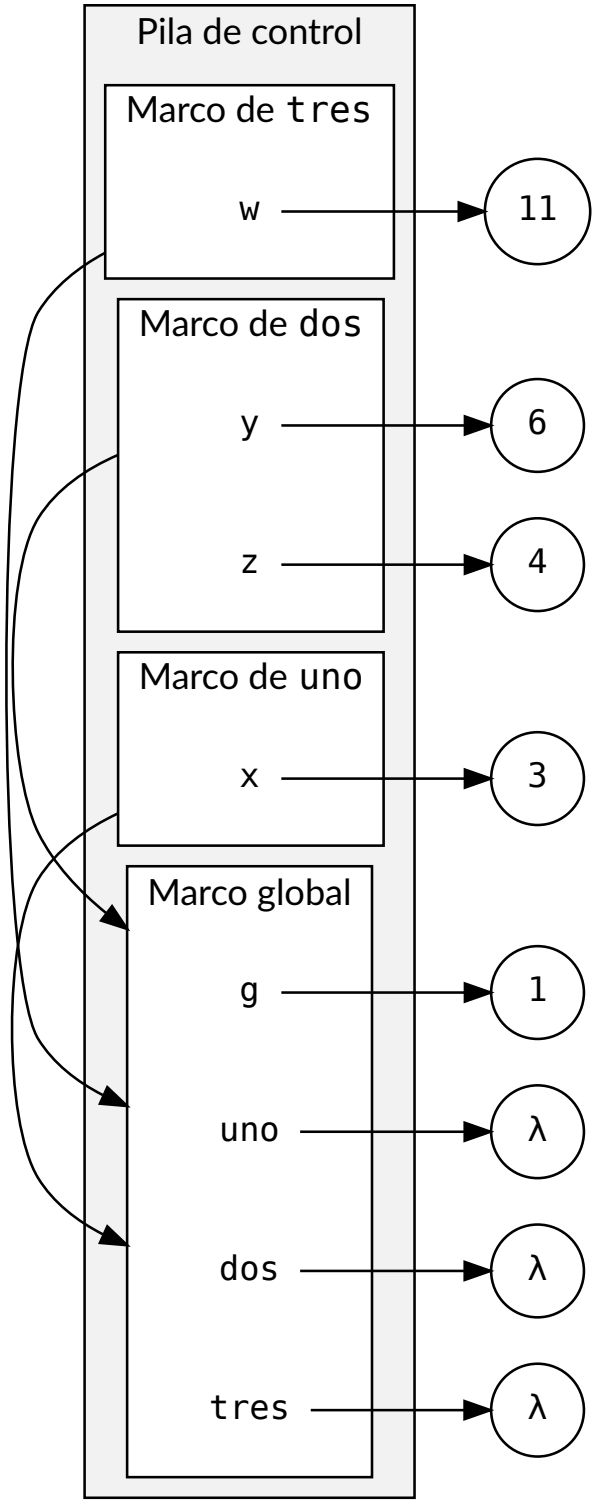
Un marco almacenado en la pila también se denomina **registro de activación**. Por tanto, también podemos decir que la pila de control almacena registros de activación.

Cada llamada activa está representada por su correspondiente marco en la pila.

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

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)
```



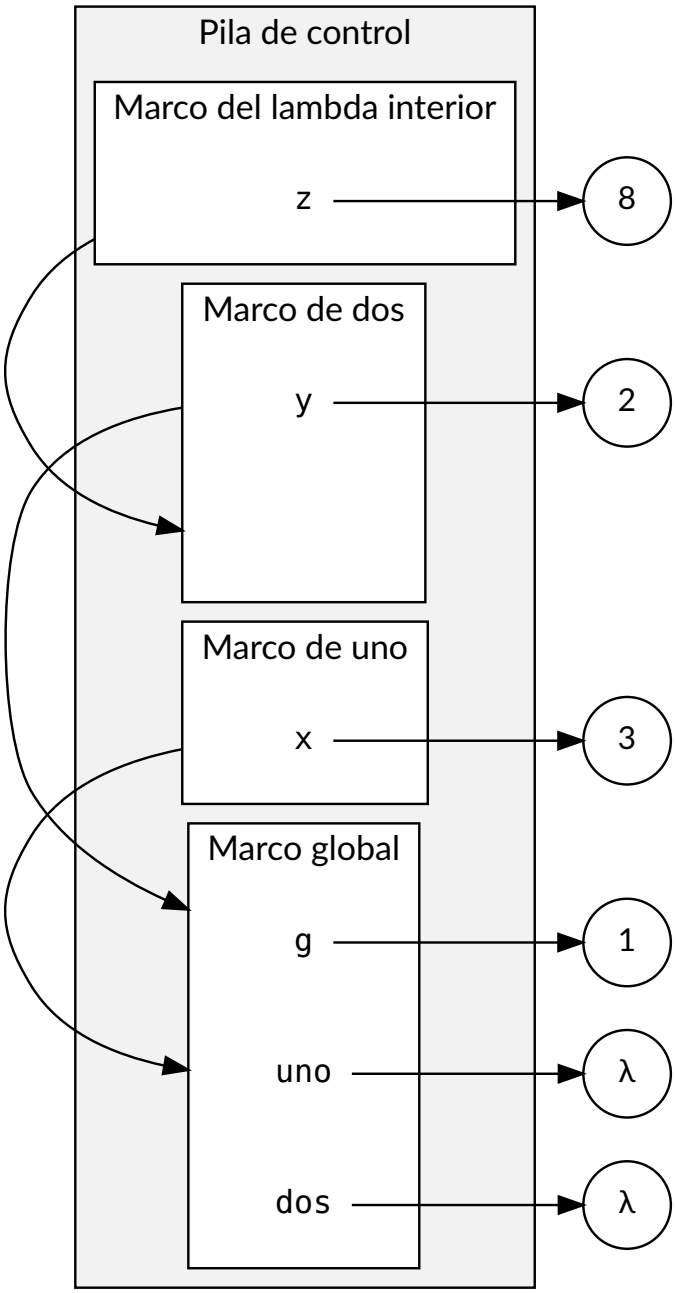
Pila de control con la llamada a la función `tres` activada

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)
```

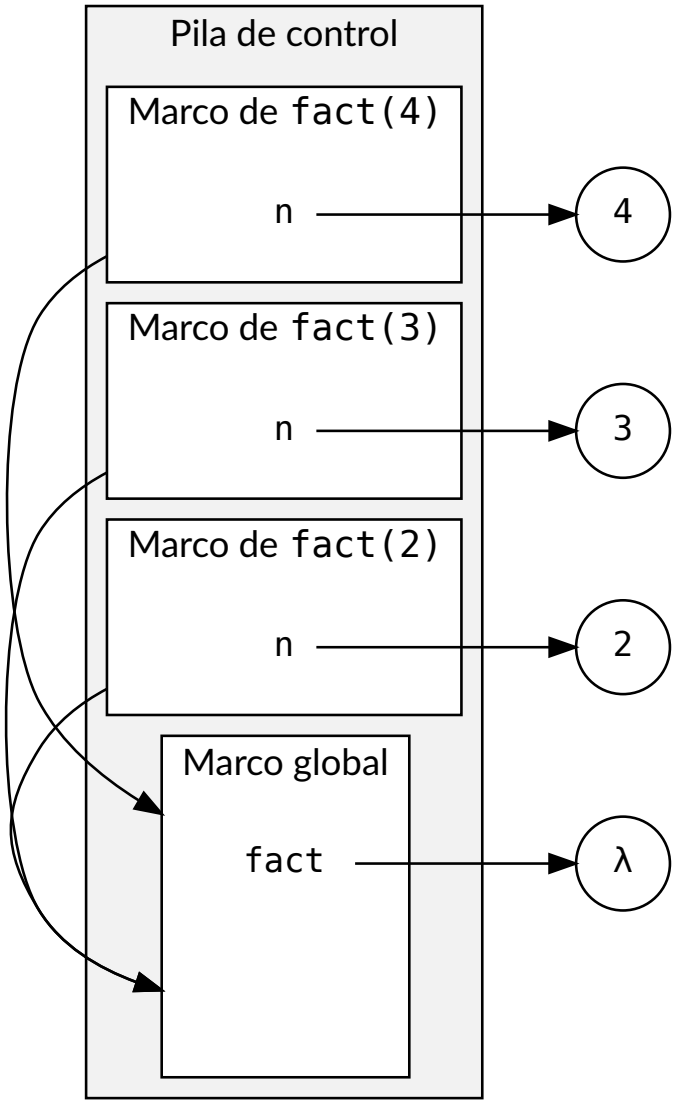


Pila de control con ámbitos anidados y la función `dos` activada

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

Por tanto, en el caso de una función recursiva, tendremos un marco 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 nueva 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.

1.5. 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.

2. Tipos de datos recursivos

2.1. Concepto

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

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

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

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

- En los casos recursivos, el tipo recursivo se define sobre sí mismo.

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

2.2. Cadenas

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

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

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

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

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

2.3. Tuplas

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

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

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

Por ejemplo:

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

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

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

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

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

* El **resto** de la tupla (al que se accede mediante `t[1:]`), que también es una tupla (*caso recursivo*).

Según el ejemplo anterior:

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

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

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

Por ejemplo:

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

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

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

2.4. Rangos

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

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

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

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

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

El valor de *stop* no se alcanza nunca.

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

step debe ser siempre distinto de cero.

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

Ejemplos

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Según el ejemplo anterior:

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

```
5
```

2.5. Conversión a tupla

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

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

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

3. Funciones de orden superior

3.1. 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.

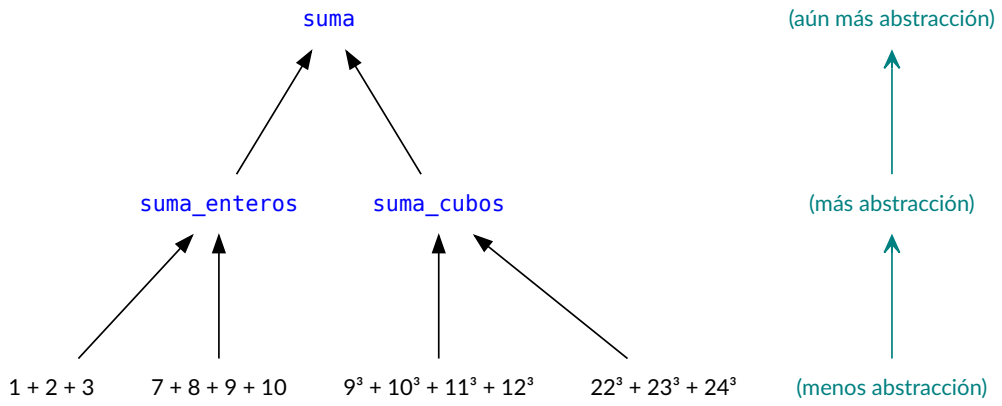
En programación funcional lo conseguimos creando funciones que conviertan los «casilleros» en parámetros que recibirían funciones:

```
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

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

3.2. `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

3. Inténtalo.

Una forma de hacerlo sería:

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

4. Inténtalo.

Podríamos definirla así:

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

3.3. `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 signatura 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)
```

3.4. 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 signatura 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 = lambda tupla: reduce(lambda x, y: x + y, tupla, 0)
producto_de_numeros =lambda tupla: reduce(lambda x, y: x * y, tupla, 1)
```

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

Ejercicio

5. Inténtalo.

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

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

3.5. 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.