Capítulo 6

Funciones

—Y ellos, naturalmente, responden a sus nombres, ¿no? —observó al desgaire el Mosquito.
—Nunca oí decir tal cosa.
—¿Pues de qué les sirve tenerlos —preguntó el Mosquito— si no responden a sus nombres?

LEWIS CARROLL, *Alicia a través del espejo*.

En capítulos anteriores hemos aprendido a utilizar funciones. Algunas de ellas están predefinidas (*abs*, *round*, etc.) mientras que otras deben importarse de módulos antes de poder ser usadas (por ejemplo, *sin* y *cos* se importan del módulo *math*). En este tema aprenderemos a definir nuestras propias funciones. Definiendo nuevas funciones estaremos «enseñando» a Python a hacer cálculos que inicialmente no sabe hacer y, en cierto modo, adaptando el lenguaje de programación al tipo de problemas que deseamos resolver, enriqueciéndolo para que el programador pueda ejecutar acciones complejas de un modo sencillo: llamando a funciones desde su programa.

Ya has usado módulos, es decir, ficheros que contienen funciones y variables de valor predefinido que puedes importar en tus programas. En este capítulo aprenderemos a crear nuestros propios módulos, de manera que reutilizar nuestras funciones en varios programas resultará extremadamente sencillo: bastará con importarlas.

6.1. Uso de funciones

Denominaremos *activar*, *invocar* o *llamar* a una función a la acción de usarla. Las funciones que hemos aprendido a invocar reciben cero, uno o más *argumentos* separados por comas y encerrados entre un par de paréntesis y pueden devolver un valor o no devolver nada.

```
>>> abs(-3) ↲
3
>>> abs(round(2.45, 1)) ↲
2.5
>>> from sys import exit ↲
>>> exit() ↲
```

Podemos llamar a una función desde una expresión. Como el resultado tiene un tipo determinado, hemos de estar atentos a que éste sea compatible con la operación y tipo de los operandos con los que se combina:

```
>>> 1 + (abs(-3) * 2) ↲
7
>>> 2.5 / abs(round(2.45, 1)) ↲
1.0
>>> 3 + str(3) ↲
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: number coercion failed
```

¿Ves? En el último caso se ha producido un error de tipos porque se ha intentado sumar una cadena, que es el tipo de dato del valor devuelto por *str*, a un entero.

Observa que los argumentos de una función también pueden ser expresiones:

```
>>> abs(round(1.0/9, 4/(1+1))) ↲
0.11
```

6.2. Definición de funciones

Vamos a estudiar el modo en que podemos definir (y usar) nuestras propias funciones Python. Estudiaremos en primer lugar cómo definir y llamar a funciones que devuelven un valor y pasaremos después a presentar los denominados procedimientos: funciones que no devuelven ningún valor. Además de los conceptos y técnicas que te iremos presentando, es interesante que te fijes en cómo desarrollamos los diferentes programas de ejemplo.

6.2.1. Definición y uso de funciones con un solo parámetro

Empezaremos definiendo una función muy sencilla, una que recibe un número y devuelve el cuadrado de dicho número. El nombre que daremos a la función es *cuadrado*. Observa este fragmento de programa:

cuadrado.py

```
1 def cuadrado(x):
2     return x ** 2
```

Ya está. Acabamos de definir la función *cuadrado* que se aplica sobre un valor al que llamamos x y devuelve un número: el resultado de elevar x al cuadrado. En el programa aparecen dos nuevas palabras reservadas: **def** y **return**. La palabra *def* es abreviatura de «define» y **return** significa «devuelve» en inglés. Podríamos leer el programa anterior como «define cuadrado de x como el valor que resulta de elevar x al cuadrado».

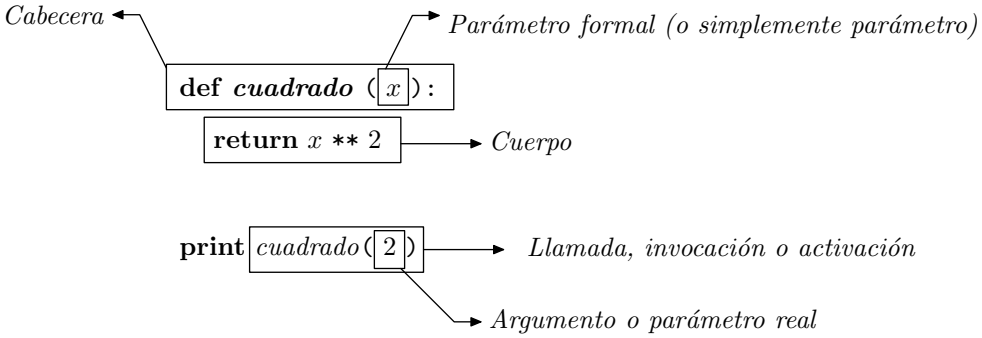
En las líneas que siguen a su definición, la función *cuadrado* puede utilizarse del mismo modo que las funciones predefinidas:

cuadrado.py

```
1 def cuadrado(x):
2     return x ** 2
3
4 print cuadrado(2)
5 a = 1 + cuadrado(3)
6 print cuadrado(a * 3)
```

En cada caso, el resultado de la expresión que sigue entre paréntesis al nombre de la función es utilizado como valor de x durante la ejecución de *cuadrado*. En la primera llamada (línea 4) el valor es 2, en la siguiente llamada es 3 y en la última, 30. Fácil, ¿no?

Detengámonos un momento para aprender algunos términos nuevos. La línea que empieza con **def** es la *cabecera* de la función y el fragmento de programa que contiene los cálculos que debe efectuar la función se denomina *cuerpo* de la función. Cuando estamos definiendo una función, su parámetro se denomina *parámetro formal* (aunque, por abreviar, normalmente usaremos el término *parámetro*, sin más). El valor que *pasamos* a una función cuando la invocamos se denomina *parámetro real* o *argumento*. Las porciones de un programa que no son cuerpo de funciones forman parte del *programa principal*: son las sentencias que se ejecutarán cuando el programa entre en acción. El cuerpo de las funciones sólo se ejecutará si se producen las correspondientes llamadas.



Definir no es invocar

Si intentamos ejecutar este programa:

cuadrado_4.py — cuadrado.py

```
1 def cuadrado(x):
2     return x ** 2
```

no ocurrirá nada en absoluto; bueno, al menos nada que aparezca por pantalla. La definición de una función sólo hace que Python «aprenda» *silenciosamente* un método de cálculo asociado al identificador *cuadrado*. Nada más. Hagamos la prueba ejecutando el programa:

```
$ python cuadrado.py ↲
```

¿Lo ves? No se ha impreso nada en pantalla. No se trata de que no haya ningún **print**, sino de que definir una función es un proceso que no tiene eco en pantalla. Repetimos: definir una función sólo asocia un método de cálculo a un identificador y no supone ejecutar dicho método de cálculo.

Este otro programa sí muestra algo por pantalla:

cuadrado_5.py — cuadrado.py

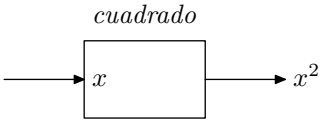
```
1 def cuadrado(x):
2     return x ** 2
3
4 print cuadrado(2)
```

Al invocar la función *cuadrado* (línea 4) se ejecuta ésta. En el programa, la invocación de la última línea provoca la ejecución de la línea 2 con un valor de x igual a 2 (argumento de la llamada). El valor devuelto con **return** es mostrado en pantalla como efecto de la sentencia **print** de la línea 4. Hagamos la prueba:

```
$ python cuadrado.py ↲
4
```

Las reglas para dar nombre a las funciones y a sus parámetros son las mismas que seguimos para dar nombre a las variables: sólo se pueden usar letras (del alfabeto inglés), dígitos y el carácter de subrayado; la primera letra del nombre no puede ser un número; y no se pueden usar palabras reservadas. Pero, ¡cuidado!: no debes dar el mismo nombre a una función y a una variable. En Python, cada nombre debe identificar claramente un único elemento: una variable o una función.[1]

Al definir una función cuadrado es como si hubiésemos creado una «máquina de calcular cuadrados». Desde la óptica de su uso, podemos representar la función como una caja que transforma un *dato de entrada* en un *dato de salida*:



[1] Más adelante, al presentar las variables locales, matizaremos esta afirmación.

Definición de funciones desde el entorno interactivo

Hemos aprendido a definir funciones dentro de un programa. También puedes definir funciones desde el entorno interactivo de Python. Te vamos a enseñar paso a paso qué ocurre en el entorno interactivo cuando estamos definiendo una función.

En primer lugar aparece el *prompt*. Podemos escribir entonces la primera línea:

```
>>> def cuadrado(x): ↲
... ↲
```

Python nos responde con tres puntos (...). Esos tres puntos son el llamado *prompt secundario*: indica que la acción de definir la función no se ha completado aún y nos pide más sentencias. Escribimos a continuación la segunda línea respetando la indentación que le corresponde:

```
>>> def cuadrado(x): ↲
...     return x ** 2 ↲
... ↲
```

Nuevamente Python responde con el *prompt* secundario. Es necesario que le demos una vez más al retorno de carro para que Python entienda que ya hemos acabado de definir la función:

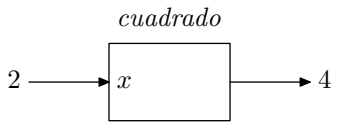
```
>>> def cuadrado(x): ↲
...     return x ** 2 ↲
... ↲
>>>
```

Ahora aparece de nuevo el *prompt* principal o primario. Python ha aprendido la función y está listo para que introduzcamos nuevas sentencias o expresiones.

```
>>> def cuadrado(x): ↲
...     return x ** 2 ↲
... ↲
>>> cuadrado(2) ↲
4
>>> 1 + cuadrado(1+3) ↲
17
>>>
```

Cuando invocas a la función, le estás «conectando» un valor a la entrada, así que la «máquina de calcular cuadrados» se pone en marcha y produce la solución deseada:

```
>>> cuadrado(2) ↲
4
```



Ojo: no hay una única forma de construir la «máquina de calcular cuadrados». Fíjate en esta definición alternativa:

cuadrado_6.py — cuadrado.py

```
1 def cuadrado(x):
2     return x * x
```

Se trata de un definición tan válida como la anterior, ni mejor, ni peor. Como usuarios de la función, poco nos importa *cómo* hace el cálculo[2]; lo que importa es *qué datos recibe* y *qué valor devuelve*.

Vamos con un ejemplo más: una función que calcula el valor de x por el seno de x:

[2] ... por el momento. Hay muchas formas de hacer el cálculo, pero unas resultan más *eficientes* (más rápidas) que otras. Naturalmente, cuando podamos elegir, escogeremos la forma más eficiente.

```
from math import sin

def xsin(x):
    return x * sin(x)
```

Lo interesante de este ejemplo es que la función definida, *xsin*, contiene una llamada a otra función (*sin*). No hay problema: desde una función puedes invocar a cualquier otra.

Una confusión frecuente

Supongamos que definimos una función con un parámetro x como esta:

```
def cubo(x):
    return x ** 3
```

Es frecuente en los aprendices confundir el parámetro x con una variable x. Así, les parece extraño que podamos invocar así a la función:

```
y = 1
print cubo(y)
```

¿Cómo es que ahora llamamos y a lo que se llamaba x? No hay problema alguno. Al definir una función, usamos un identificador cualquiera para referirnos al parámetro. Tanto da que se llame x como y. Esta otra definición de cubo es absolutamente equivalente:

```
def cubo(z):
    return z ** 3
```

La definición se puede leer así: «si te pasan un valor, digamos z, devuelve ese valor elevado al cubo». Usamos el nombre z (o x) sólo para poder referirnos a él en el cuerpo de la función.

EJERCICIOS

▶ **262** Define una función llamada *raiz_cubica* que devuelva el valor de $\sqrt[3]{x}$.
(Nota: recuerda que $\sqrt[3]{x}$ es $x^{1/3}$ y ándate con ojo, no sea que utilices una división entera y eleves x a la potencia 0, que es el resultado de calcular 1/3.)

▶ **263** Define una función llamada *area_circulo* que, a partir del radio de un círculo, devuelva el valor de su área. Utiliza el valor 3.1416 como aproximación de π o importa el valor de π que encontrarás en el módulo *math*.
(Recuerda que el área de un círculo es πr^2.)

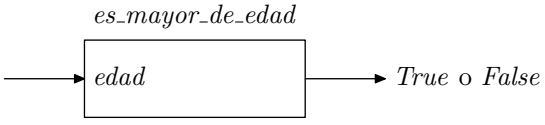
▶ **264** Define una función que convierta grados Farenheit en grados centígrados.
(Para calcular los grados centígrados has de restar 32 a los grados Farenheit y multiplicar el resultado por cinco novenos.)

▶ **265** Define una función que convierta grados centígrados en grados Farenheit.

▶ **266** Define una función que convierta radianes en grados.
(Recuerda que 360 grados son 2π radianes.)

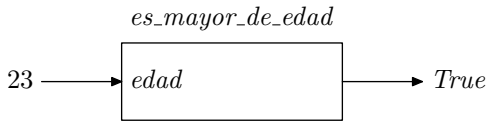
▶ **267** Define una función que convierta grados en radianes.

En el cuerpo de una función no sólo pueden aparecer sentencias **return**; también podemos usar estructuras de control: sentencias condicionales, bucles, etc. Lo podemos comprobar diseñando una función que recibe un número y devuelve un booleano. El valor de entrada es la edad de una persona y la función devuelve *True* si la persona es mayor de edad y *False* en caso contrario:

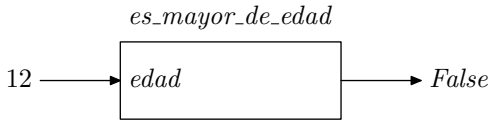


Cuando llamas a la función, ésta se activa para producir un resultado concreto (en nuestro caso, o bien devuelve *True* o bien devuelve *False*):

```
a = es_mayor_de_edad(23)
```




```
b = es_mayor_de_edad(12)
```



Una forma usual de devolver valores de función es a través de un sólo **return** ubicado al final del cuerpo de la función:

mayoria_edad_4.py — mayoria_edad.py

```
def es_mayor_de_edad(edad):
    if edad < 18:
        resultado = False
    else:
        resultado = True
    return resultado
```

Pero no es el único modo en que puedes devolver diferentes valores. Mira esta otra definición de la misma función:

mayoria_edad.py — mayoria_edad.py

```
def es_mayor_de_edad(edad):
    if edad < 18:
        return False
    else:
        return True
```

Aparecen dos sentencias **return**: cuando la ejecución llega a cualquiera de ellas, finaliza *inmediatamente* la llamada a la función y se devuelve el valor que sigue al **return**. Podemos asimilar el comportamiento de **return** al de **break**: una sentencia **break** fuerza a terminar la ejecución de un bucle y una sentencia **return** fuerza a terminar la ejecución de una llamada a función.

EJERCICIOS

▶ **268** ¿Es este programa equivalente al que acabamos de ver?

mayoria_edad_5.py — mayoria_edad.py

```
def mayoria_de_edad(edad):
    if edad < 18:
        return False
    return True
```

▶ **269** ¿Es este programa equivalente al que acabamos de ver?

mayoria_edad_6.py — mayoria_edad.py

```
def mayoria_de_edad(edad):
    return edad >= 18
```

▶ **270** La última letra del DNI puede calcularse a partir del número. Para ello sólo tienes que dividir el número por 23 y quedarte con el resto, que es un número entre 0 y 22. La letra que corresponde a cada número la tienes en esta tabla:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

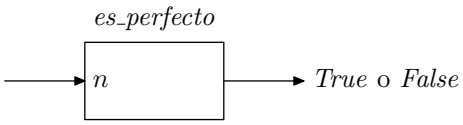
Define una función que, dado un número de DNI, devuelva la letra que le corresponde.

▶ **271** Diseña una función que reciba una cadena y devuelva cierto si empieza por minúscula y falso en caso contrario.

▶ **272** Diseña una función llamada *es_repeticion* que reciba una cadena y nos diga si la cadena está formada mediante la concatenación de una cadena consigo misma. Por ejemplo, *es_repeticion*`('abab')` devolverá *True*, pues la cadena `'abab'` está formada con la cadena `'ab'` repetida; por contra *es_repeticion*`('ababab')` devolverá *False*.

Y ahora, un problema más complicado. Vamos a diseñar una función que nos diga si un número dado es o no es *perfecto*. Se dice que un número es perfecto si es igual a la suma de todos sus divisores excluído él mismo. Por ejemplo, 28 es un número perfecto, pues sus divisores (excepto él mismo) son 1, 2, 4, 7 y 14, que suman 28.

Empecemos. La función, a la que llamaremos *es_perfecto* recibirá un sólo dato (el número sobre el que hacemos la pregunta) y devolverá un valor booleano:



La cabecera de la función está clara:

perfecto.py
```
1 def es_perfecto(n):
2     ...
```

¿Y por dónde seguimos? Vamos por partes. En primer lugar estamos interesados en conocer todos los divisores del número. Una vez tengamos claro cómo saber cuáles son, los sumaremos. Si la suma coincide con el número original, éste es perfecto; si no, no. Podemos usar un bucle y preguntar a todos los números entre 1 y n−1 si son divisores de n:

perfecto.py
```
1 def es_perfecto(n):
2     for i in range(1, n):
3         if i es divisor de n:
4             ...
```

Observa cómo seguimos siempre la reglas de indentación de código que impone Python. ¿Y cómo preguntamos ahora si un número es divisor de otro? El operador módulo `%` devuelve el resto de la división y resuelve fácilmente la cuestión:

perfecto.py
```
1 def es_perfecto(n):
2     for i in range(1, n):
3         if n % i == 0:
4             ...
```

La línea 4 sólo se ejecutará para valores de i que son divisores de n. ¿Qué hemos de hacer a continuación? Deseamos sumar todos los divisores y ya conocemos la «plantilla» para calcular sumatorios:

perfecto.py
```
1 def es_perfecto(n):
2     sumatorio = 0
3     for i in range(1, n):
4         if n % i == 0:
5             sumatorio += i
6     ...
```

¿Qué queda por hacer? Comprobar si el número es perfecto y devolver *True* o *False*, según proceda:

perfecto_3.py

perfecto.py
```
1 def es_perfecto(n):
2     sumatorio = 0
```

```
    for i in range(1, n):
        if n % i == 0:
            sumatorio += i
    if sumatorio == n:
        return True
    else:
        return False
```

Y ya está. Bueno, podemos simplificar un poco las cuatro últimas líneas y convertirlas en una sola. Observa esta nueva versión:

perfecto.py — perfecto.py

```
def es_perfecto(n):
    sumatorio = 0
    for i in range(1, n):
        if n % i == 0:
            sumatorio += i
    return sumatorio == n
```

¿Qué hace la última línea? Devuelve el resultado de evaluar la expresión lógica que compara *sumatorio* con *n*: si ambos números son iguales, devuelve *True*, y si no, devuelve *False*. Mejor, ¿no?

EJERCICIOS

▶ **273** ¿En qué se ha equivocado nuestro aprendiz de programador al escribir esta función?

perfecto_4.py — perfecto.py

```
def es_perfecto(n):
    for i in range(1, n):
        sumatorio = 0
        if n % i == 0:
            sumatorio += i
    return sumatorio == n
```

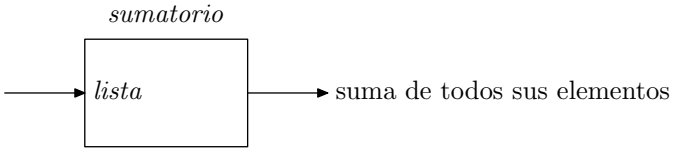
▶ **274** Mejora la función *es_perfecto* haciéndola más rápida. ¿Es realmente necesario considerar todos los números entre 1 y n-1?

▶ **275** Diseña una función que devuelva una lista con los números perfectos comprendidos entre 1 y n, siendo n un entero que nos proporciona el usuario.

▶ **276** Define una función que devuelva el número de días que tiene un año determinado. Ten en cuenta que un año es bisiesto si es divisible por 4 y no divisible por 100, excepto si es también divisible por 400, en cuyo caso es bisiesto.

(Ejemplos: El número de días de 2002 es 365: el número 2002 no es divisible por 4, así que no es bisiesto. El año 2004 es bisiesto y tiene 366 días: el número 2004 es divisible por 4, pero no por 100, así que es bisiesto. El año 1900 es divisible por 4, pero no es bisiesto porque es divisible por 100 y no por 400. El año 2000 sí es bisiesto: el número 2000 es divisible por 4 y, aunque es divisible por 100, también lo es por 400.)

Hasta el momento nos hemos limitado a suministrar valores escalares como argumentos de una función, pero también es posible suministrar argumentos de tipo secuencial. Veámoslo con un ejemplo: una función que recibe una lista de números y nos devuelve el sumatorio de todos sus elementos.



suma_lista_4.py — suma_lista.py

```
def sumatorio(lista):
    s = 0
    for numero in lista:
        s += numero
    return s
```

Podemos usar la función así:

suma_lista_5.py suma_lista.py

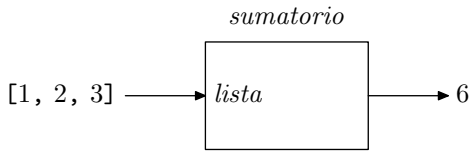
```
...
a = [1, 2, 3]
print sumatorio(a)
```

o así:

suma_lista_6.py suma_lista.py

```
...
print sumatorio([1, 2, 3])
```

En cualquiera de los dos casos, el parámetro *lista* toma el valor `[1, 2, 3]`, que es el argumento suministrado en la llamada:



Sumatorios

Has aprendido a calcular sumatorios con bucles. Desde la versión 2.3, Python ofrece una forma mucho más cómoda de calcular sumatorios: la función predefinida *sum*, que recibe una lista de valores y devuelve el resultado de sumarlos.

```
>>> sum([1, 10, 20]) ↲
31
```

¿Cómo usarla para calcular el sumatorio de los 100 primeros números naturales? Muy fácil: pasándole una lista con esos número, algo que resulta trivial si usas *range*.

```
>>> sum(range(101)) ↲
5050
```

Mmmm. Ten cuidado: *range* construye una lista en memoria. Si calculas así el sumatorio del primer millón de números es posible que te quedes sin memoria. Hay una función alternativa, *xrange*, que no construye la lista en memoria, pero que hace creer a quien la recorre que es una lista en memoria:

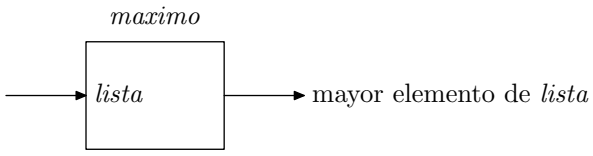
```
>>> sum(xrange(1000001)) ↲
500000500000L
```

EJERCICIOS

▶ **277** Diseña una función que calcule el sumatorio de la diferencia entre números contiguos en una lista. Por ejemplo, para la lista `[1, 3, 6, 10]` devolverá 9, que es $2+3+4$ (el 2 resulta de calcular $3-1$, el 3 de calcular $6-3$ y el 4 de calcular $10-6$).

¿Sabes efectuar el cálculo de ese sumatorio sin utilizar bucles (ni la función *sum*)?

Estudiemos otro ejemplo: una función que recibe una lista de números y devuelve el valor de su mayor elemento.



La idea básica es sencilla: recorrer la lista e ir actualizando el valor de una variable auxiliar que, en todo momento, contendrá el máximo valor visto hasta ese momento.

maximo_7.py — maximo.py

```
1 def maximo(lista):
2     for elemento in lista:
3         if elemento > candidato:
4             candidato = elemento
5     return candidato
```

Nos falta inicializar la variable *candidato*. ¿Con qué valor? Podríamos pensar en inicializarla con el menor valor posible. De ese modo, cualquier valor de la lista será mayor que él y es seguro que su valor se modificará tan pronto empecemos a recorrer la lista. Pero hay un problema: no sabemos cuál es el menor valor posible. Una buena alternativa es inicializar *candidato* con el valor del primer elemento de la lista. Si ya es el máximo, perfecto, y si no lo es, más tarde se modificará *candidato*.

maximo_8.py — maximo.py

```
1 def maximo(lista):
2     candidato = lista[0]
3     for elemento in lista:
4         if elemento > candidato:
5             candidato = elemento
6     return candidato
```

EJERCICIOS

▶ **278** Haz una traza de la llamada *maximo*([6, 2, 7, 1, 10, 1, 0]).

¿Ya está? Aún no. ¿Qué pasa si se proporciona una lista vacía como entrada? La línea 2 provocará un error de tipo *IndexError*, pues en ella intentamos acceder al primer elemento de la lista... y la lista vacía no tiene ningún elemento. Un objetivo es, pues, evitar ese error. Pero, en cualquier caso, algo hemos de devolver como máximo elemento de una lista, ¿y qué valor podemos devolvemos como máximo elemento de una lista vacía? Mmmm. A bote pronto, tenemos dos posibilidades:

- Devolver un valor especial, como el valor 0. Mejor no. Tiene un serio inconveniente: ¿cómo distinguiré el máximo de [-3, -5, 0, -4], que es un cero «legítimo», del máximo de []?
- O devolver un valor «muy» especial, como el valor *None*. ¿Que qué es *None*? *None* significa en inglés «ninguno» y es un valor predefinido en Python que se usa para denotar «ausencia de valor». Como el máximo de una lista vacía no existe, parece acertado devolver la «ausencia de valor» como máximo de sus miembros.

Nos inclinamos por esta segunda opción. En adelante, usaremos *None* siempre que queramos referirnos a un valor «muy» especial: a la ausencia de valor.

maximo.py

```
1 def maximo(lista):
2     if len(lista) > 0:
3         candidato = lista[0]
4         for elemento in lista:
5             if elemento > candidato:
6                 candidato = elemento
7     else:
8         candidato = None
9     return candidato
```

EJERCICIOS

▶ **279** Diseña una función que, dada una lista de números enteros, devuelva el número de «series» que hay en ella. Llamamos «serie» a todo tramo de la lista con valores idénticos.

Por ejemplo, la lista [1, 1, 8, 8, 8, 8, 0, 0, 0, 2, 10, 10] tiene 5 «series» (ten en cuenta que el 2 forma parte de una «serie» de un solo elemento).

▶ **280** Diseña una función que diga en qué posición empieza la «serie» más larga de una lista. En el ejemplo del ejercicio anterior, la «serie» más larga empieza en la posición 2 (que es el índice donde aparece el primer 8). (Nota: si hay dos «series» de igual longitud y ésta es la mayor, debes devolver la posición de la primera de las «series». Por ejemplo, para [8, 2, 2, 9, 9] deberás devolver la posición 1.)

▶ **281** Haz una función que reciba una lista de números y devuelva la media de dichos números. Ten cuidado con la lista vacía (su media es cero).

▶ **282** Diseña una función que calcule el productorio de todos los números que componen una lista.

▶ **283** Diseña una función que devuelva el valor absoluto de la máxima diferencia entre dos elementos consecutivos de una lista. Por ejemplo, el valor devuelto para la lista `[1, 10, 2, 6, 2, 0]` es 9, pues es la diferencia entre el valor 1 y el valor 10.

▶ **284** Diseña una función que devuelva el valor absoluto de la máxima diferencia entre cualquier par de elementos de una lista. Por ejemplo, el valor devuelto para la lista `[1, 10, 2, 6, 8, 2 0]` es 9, pues es la diferencia entre el valor 10 y el valor 0. (Pista: te puede convenir conocer el valor máximo y el valor mínimo de la lista.)

▶ **285** Modifica la función del ejercicio anterior para que devuelva el valor 0 tan pronto encuentre un 0 en la lista.

▶ **286** Define una función que, dada una cadena x, devuelva otra cuyo contenido sea el resultado de concatenar 6 veces x consigo misma.

▶ **287** Diseña una función que, dada una lista de cadenas, devuelva la cadena más larga. Si dos o más cadenas miden lo mismo y son las más largas, la función devolverá una cualquiera de ellas.

(Ejemplo: dada la lista `['Pepe', 'Juan', 'María', 'Ana']`, la función devolverá la cadena `'María'`.)

▶ **288** Diseña una función que, dada una lista de cadenas, devuelva *una lista con todas* las cadenas más largas, es decir, si dos o más cadenas miden lo mismo y son las más largas, la lista las contendrá a todas.

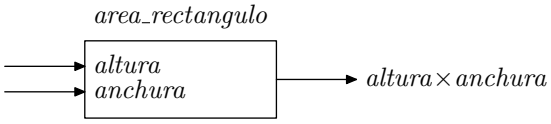
(Ejemplo: dada la lista `['Pepe', 'Ana', 'Juan', 'Paz']`, la función devolverá la lista de dos elementos `['Pepe', 'Juan']`.)

▶ **289** Diseña una función que reciba una lista de cadenas y devuelva el prefijo común más largo. Por ejemplo, la cadena `'pol'` es el prefijo común más largo de esta lista:

```
['poliedro', 'policía', 'polífona', 'polinizar', 'polaridad', 'política']
```

6.2.2. Definición y uso de funciones con varios parámetros

No todas las funciones tienen un sólo parámetro. Vamos a definir ahora una con dos parámetros: una función que devuelve el valor del área de un rectángulo dadas su altura y su anchura:



rectangulo.py

```
def area_rectangulo(altura, anchura):
    return altura * anchura
```

Observa que los diferentes parámetros de una función deben separarse por comas. Al usar la función, los argumentos también deben separarse por comas:

rectangulo_2.py rectangulo.py

```
def area_rectangulo(altura, anchura):
    return altura * anchura

print area_rectangulo(3, 4)
```

Importaciones, definiciones de función y programa principal

Los programas que diseñes a partir de ahora tendrán tres «tipos de línea»: importación de módulos (o funciones y variables de módulos), definición de funciones y sentencias del programa principal. En principio puedes alternar líneas de los tres tipos. Mira este programa, por ejemplo,

```
def cuadrado(x):
    return x**2

vector = []
for i in range(3):
    vector.append(float(raw_input('Dame un número: ')))

def suma_cuadrados(v):
    s = 0
    for e in v:
        s += cuadrado(e)
    return s

y = suma_cuadrados(vector)

from math import sqrt

print 'Distancia al origen:', sqrt(y)
```

En él se alternan definiciones de función, importaciones de funciones y sentencias del programa principal, así que resulta difícil hacerse una idea clara de qué hace el programa. No diseñes así tus programas.

Esta otra versión del programa anterior pone en primer lugar las importaciones, a continuación, las funciones y, al final, de un tirón, las sentencias que conforman el programa principal:

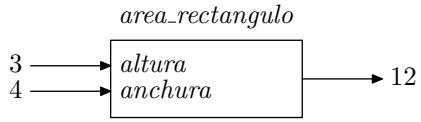
```
from math import sqrt

def cuadrado(x):
    return x**2

def suma_cuadrados(v):
    s = 0
    for e in v:
        s += cuadrado(e)
    return s

# Programa principal
vector = []
for i in range(3):
    vector.append(float(raw_input('Dame un número: ')))
y = suma_cuadrados(vector)
print 'Distancia al origen:', sqrt(y)
```

Es mucho más legible. Te recomendamos que sigas siempre esta organización en tus programas. Recuerda que la legibilidad de los programas es uno de los objetivos del programador.



EJERCICIOS

▶ **290** Define una función que, dado el valor de los tres lados de un triángulo, devuelva la longitud de su perímetro.

▶ **291** Define una función que, dados dos parámetros b y x, devuelva el valor de $\log_b(x)$, es decir, el logaritmo en base b de x.

▶ **292** Diseña una función que devuelva la solución de la ecuación lineal $ax + b = 0$ dados a y b. Si la ecuación tiene infinitas soluciones o no tiene solución alguna, la función lo detectará y devolverá el valor *None*.

▶ **293** Diseña una función que calcule $\sum_{i=a}^{b} i$ dados a y b. Si a es mayor que b, la función devolverá el valor 0.

▶ **294** Diseña una función que calcule $\prod_{i=a}^{b} i$ dados a y b. Si a es mayor que b, la función devolverá el valor 0. Si 0 se encuentra entre a y b, la función devolverá también el valor cero, pero sin necesidad de iterar en un bucle.

▶ **295** Define una función llamada *raiz_n_esima* que devuelva el valor de $\sqrt[n]{x}$. (Nota: recuerda que $\sqrt[n]{x}$ es $x^{1/n}$).

▶ **296** Haz una función que reciba un número de DNI y una letra. La función devolverá *True* si la letra corresponde a ese número de DNI, y *False* en caso contrario. La función debe llamarse *comprueba_letra_dni*.

Si lo deseas, puedes llamar a la función *letra_dni*, desarrollada en el ejercicio 270, desde esta nueva función.

▶ **297** Diseña una función que diga (mediante la devolución de *True* o *False*) si dos números son *amigos*. Dos números son amigos si la suma de los divisores del primero (excluído él) es igual al segundo y viceversa.

6.2.3. Definición y uso de funciones sin parámetros

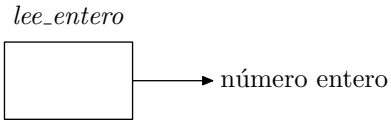
Vamos a considerar ahora cómo definir e invocar funciones sin parámetros. En realidad hay poco que decir: lo único que debes tener presente es que es obligatorio poner paréntesis a continuación del identificador, tanto al definir la función como al invocarla.

En el siguiente ejemplo se define y usa una función que lee de teclado un número entero:

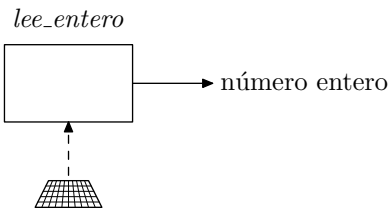
lee_entero.py

```
1 def lee_entero():
2     return int(raw_input())
3
4 a = lee_entero()
```

Recuerda: al llamar a una función los paréntesis *no* son opcionales. Podemos representar esta función como una caja que proporciona un dato de salida sin ningún dato de entrada:



Mmmm. Te hemos dicho que la función no recibe dato alguno y debes estar pensando que te hemos engañado, pues la función lee un dato de teclado. Quizá este diagrama represente mejor la entrada/salida función:



De acuerdo; pero no te equivoques: el dato leído de teclado no es un dato que el programa suministre a la función.

Esta otra función lee un número de teclado y se asegura de que sea positivo:

Parámetros o teclado

Un error frecuente al diseñar funciones consiste en tratar de obtener la información directamente de teclado. No es que esté prohibido, pero es ciertamente excepcional que una función obtenga la información de ese modo. Cuando te pidan diseñar una función que recibe uno o más datos, se sobreentiende que debes suministrarlos como argumentos en la llamada, no leerlos de teclado. Cuando queramos que la función lea algo de teclado, lo diremos *explícitamente*.

Insistimos y esta vez ilustrando el error con un ejemplo. Imagina que te piden que diseñes una función que diga si un número es par devolviendo *True* si es así y *False* en caso contrario. Te piden una función como ésta:

```
def es_par(n):
    return n % 2 == 0
```

Muchos programadores novatos escriben *erróneamente* una función como esta otra:

```
def es_par():
    n = int(raw_input('Dame un número: '))
    return n % 2 == 0
```

Está mal. Escribir esa función así demuestra, cuando menos, falta de soltura en el diseño de funciones. Si hubiésemos querido una función como ésa, te hubiésemos pedido una función que lea de teclado un número entero y devuelva *True* si es par y *False* en caso contrario.

lee_positivo_2.py — lee_positivo.py

```
def lee_entero_positivo():
    numero = int(raw_input())
    while numero < 0:
        numero = int(raw_input())
    return numero

a = lee_entero_positivo()
```

Y esta versión muestra por pantalla un mensaje informativo cuando el usuario se equivoca:

lee_positivo.py — lee_positivo.py

```
def lee_entero_positivo():
    numero = int(raw_input())
    while numero < 0:
        print 'Ha cometido un error: el número debe ser positivo.'
        numero = int(raw_input())
    return numero

a = lee_entero_positivo()
```

Una posible aplicación de la definición de funciones sin argumentos es la presentación de menús con selección de opción por teclado. Esta función, por ejemplo, muestra un menú con tres opciones, pide al usuario que seleccione una y se asegura de que la opción seleccionada es válida. Si el usuario se equivoca, se le informa por pantalla del error:

funcion_menu.py — funcion_menu.py

```
def menu():
    opcion = ''
    while not ('a' <= opcion <= 'c'):
        print 'Cajero automático.'
        print 'a) Ingresar dinero.'
        print 'b) Sacar dinero.'
        print 'c) Consultar saldo.'
        opcion = raw_input('Escoja una opción: ')
        if not (opcion >= 'a' and opcion <= 'c'):
            print 'Sólo puede escoger las letras a, b o c. Inténtelo de nuevo.'
    return opcion
```

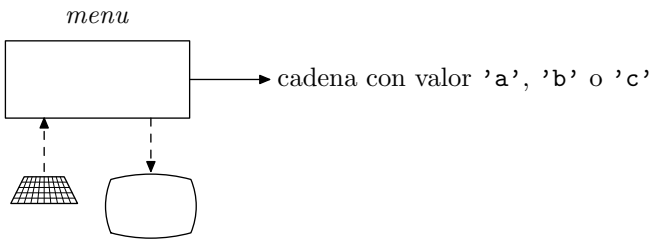
Los paréntesis son necesarios

Un error típico de los aprendices es llamar a las funciones sin parámetros omitiendo los paréntesis, pues les parecen innecesarios. Veamos qué ocurre en tal caso:

```
>>> def saluda(): ↲
...     print 'Hola' ↲
... ↲
>>> saluda() ↲
Hola
>>> saluda ↲
<function saluda at 0x8160854>
```

Como puedes ver, el último resultado no es la impresión del mensaje «`Hola`», sino otro encerrado entre símbolos de menor y mayor. Estamos llamando incorrectamente a la función: *saluda*, sin paréntesis, es un «objeto» Python ubicado en la dirección de memoria 8160854 en hexadecimal (número que puede ser distinto con cada ejecución).

Ciertas técnicas avanzadas de programación sacan partido del uso del identificador de la función sin paréntesis, pero aún no estás preparado para entender cómo y por qué. El cuadro «Un método de integración genérico» (página 269) te proporcionará más información.



Hemos dibujado una pantalla para dejar claro que uno de los cometidos de la esta función es mostrar información por pantalla (las opciones del menú).

Si en nuestro programa principal se usa con frecuencia el menú, bastará con efectuar las correspondientes llamadas a la función *menu*() y almacenar la opción seleccionada en una variable. Así:

```
accion = menu()
```

La variable *accion* contendrá la letra seleccionada por el usuario. Gracias al control que efectúa la función, estaremos seguros de que dicha variable contiene una `'a'`, una `'b'` o una `'c'`.

EJERCICIOS

▶ **298** ¿Funciona esta otra versión de *menu*?

funcion_menu_2.py

```
funcion_menu.py
1  def menu():
2      opcion = ''
3      while len(opcion) != 1 or opcion not in 'abc':
4          print 'Cajero automático.'
5          print 'a) Ingresar dinero.'
6          print 'b) Sacar dinero.'
7          print 'c) Consultar saldo.'
8          opcion = raw_input('Escoja una opción: ')
9          if len(opcion) != 1 or opcion not in 'abc':
10             print 'Sólo puede escoger las letras a, b o c. Inténtelo de nuevo.'
11     return opcion
```

▶ **299** Diseña una función llamada *menu_generico* que reciba una lista con opciones. Cada opción se asociará a un número entre 1 y la talla de la lista y la función mostrará por pantalla el menú con el número asociado a cada opción. El usuario deberá introducir por teclado una opción. Si la opción es válida, se devolverá su valor, y si no, se le advertirá del error y se solicitará nuevamente la introducción de un valor.

He aquí un ejemplo de llamada a la función:

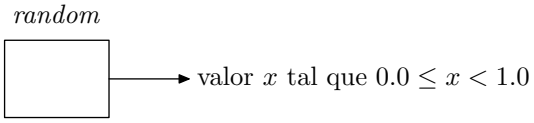
menu_generico(['Saludar', 'Despedirse', 'Salir'])

Al ejecutarla, obtendremos en pantalla el siguiente texto:

```
1) Saludar
2) Despedirse
3) Salir
Escoja opción:
```

▶ **300** En un programa que estamos diseñando preguntamos al usuario numerosas cuestiones que requieren una respuesta afirmativa o negativa. Diseña una función llamada *si_o_no* que reciba una cadena (la pregunta). Dicha cadena se mostrará por pantalla y se solicitará al usuario que responda. Sólo aceptaremos como respuestas válidas `'si'`, `'s'`, `'Si'`, `'SI'`, `'no'`, `'n'`, `'No'`, `'NO'`, las cuatro primeras para respuestas afirmativas y las cuatro últimas para respuestas negativas. Cada vez que el usuario se equivoque, en pantalla aparecerá un mensaje que le recuerde las respuestas aceptables. La función devolverá *True* si la respuesta es afirmativa, y *False* en caso contrario.

Hay funciones sin parámetros que puedes importar de módulos. Una que usaremos en varias ocasiones es *random* (en inglés «random» significa «aleatorio»). La función *random*, definida en el módulo que tiene el mismo nombre, devuelve un número al azar mayor o igual que 0.0 y menor que 1.0.



Veamos un ejemplo de uso de la función:

```
>>> from random import random ↲
>>> random() ↲
0.73646697433706487
>>> random() ↲
0.6416606281483086
>>> random() ↲
0.36339080016840919
>>> random() ↲
0.99622235710683393
```

¿Ves? La función se invoca sin argumentos (entre los paréntesis no hay nada) y cada vez que lo hacemos obtenemos un resultado diferente. ¿Qué interés tiene una función tan extraña? Una función capaz de generar números aleatorios encuentra muchos campos de aplicación: estadística, videojuegos, simulación, etc. Dentro de poco le sacaremos partido.

EJERCICIOS

▶ **301** Diseña una función sin argumentos que devuelva un número aleatorio mayor o igual que 0.0 y menor que 10.0. Puedes llamar a la función *random* desde tu función.

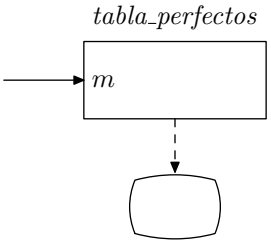
▶ **302** Diseña una función sin argumentos que devuelva un número aleatorio mayor o igual que -10.0 y menor que 10.0.

▶ **303** Para diseñar un juego de tablero nos vendrá bien disponer de un «dado electrónico». Escribe una función Python sin argumentos llamada *dado* que devuelva un número *entero* aleatorio entre 1 y 6.

6.2.4. Procedimientos: funciones sin devolución de valor

No todas las funciones devuelven un valor. Una función que no devuelve un valor se denomina *procedimiento*. ¿Y para qué sirve una función que no devuelve nada? Bueno, puede, por ejemplo, mostrar mensajes o resultados por pantalla. No te equivoques: *mostrar* algo por pantalla no es *devolver* nada. Mostrar un mensaje por pantalla es un *efecto secundario*.

Veámoslo con un ejemplo. Vamos a implementar ahora un programa que solicita al usuario un número y muestra por pantalla todos los números perfectos entre 1 y dicho número.



Reutilizaremos la función *es_perfecto* que definimos antes en este mismo capítulo. Como la solución no es muy complicada, te la ofrecemos completamente desarrollada:

tabla_perfectos_2.py — tabla_perfectos.py

```
1  def es_perfecto(n): # Averigua si el número n es o no es perfecto.
2      sumatorio = 0
3      for i in range(1, n):
4          if n % i == 0:
5              sumatorio += i
6      return sumatorio == n
7
8  def tabla_perfectos(m): # Muestra todos los números perfectos entre 1 y m.
9      for i in range(1, m+1):
10         if es_perfecto(i):
11             print i, 'es un número perfecto'
12
13 numero = int(raw_input('Dame un número: '))
14 tabla_perfectos(numero)
```

Fíjate en que la función *tabla_perfectos* no devuelve nada (no hay sentencia **return**): es un procedimiento. También resulta interesante la línea 10: como *es_perfecto* devuelve *True* o *False*, podemos utilizarla directamente como condición del **if**.

EJERCICIOS

▶ **304** Diseña un programa que, dado un número n, muestre por pantalla todas las parejas de números amigos menores que n. La impresión de los resultados debe hacerse desde un procedimiento.

Dos números amigos sólo deberán aparecer una vez por pantalla. Por ejemplo, 220 y 284 son amigos: si aparece el mensaje «`220 y 284 son amigos`», no podrá aparecer el mensaje «`284 y 220 son amigos`», pues es redundante.

Debes diseñar una función que diga si dos números son amigos y un procedimiento que muestre la tabla.

▶ **305** Implementa un procedimiento Python tal que, dado un número entero, muestre por pantalla sus cifras en orden inverso. Por ejemplo, si el procedimiento recibe el número 324, mostrará por pantalla el 4, el 2 y el 3 (en líneas diferentes).

▶ **306** Diseña una función *es_primo* que determine si un número es primo (devolviendo *True*) o no (devolviendo *False*). Diseña a continuación un procedimiento *muestra_primos* que reciba un número y muestre por pantalla todos los números primos entre 1 y dicho número.

¿Y qué ocurre si utilizamos un procedimiento como si fuera una función con devolución de valor? Podemos hacer la prueba. Asignemos a una variable el resultado de llamar a *tabla_perfectos* y mostremos por pantalla el valor de la variable:

tabla_perfectos.py — tabla_perfectos.py

```
⋮

12
13 numero = int(raw_input('Dame un número: '))
14 resultado = tabla_perfectos(100)
15 print resultado
```

Por pantalla aparece lo siguiente:

Condicionales que trabajan directamente con valores lógicos

Ciertas funciones devuelven directamente un valor lógico. Considera, por ejemplo, esta función, que nos dice si un número es o no es par:

```
def es_par(n):
    return n % 2 == 0
```

Si una sentencia condicional toma una decisión en función de si un número es par o no, puedes codificar así la condición:

```
if es_par(n):
    ...
```

Observa que no hemos usado comparador alguno en la condición del **if**. ¿Por qué? Porque la función *es_par*(*n*) devuelve *True* o *False* directamente. Los programadores primerizos tienen tendencia a codificar la misma condición así:

```
if es_par(n) == True:
    ...
```

Es decir, comparan el valor devuelto por *es_par* con el valor *True*, pues les da la sensación de que un **if** sin comparación no está completo. No pasa nada si usas la comparación, pero es innecesaria. Es más, si no usas la comparación, el programa es más legible: la sentencia condicional se lee directamente como «si *n* es par» en lugar de «si *n* es par es cierto», que es un extraño circunloquio.

Si en la sentencia condicional se desea comprobar que el número es impar, puedes hacerlo así:

```
if not es_par(n):
    ...
```

Es muy legible: «si no es par *n*».

Nuevamente, los programadores que están empezando escriben:

```
if es_par(n) == False:
    ...
```

que se lee como «si *n* es par es falso». Peor, ¿no?

Acostúmbrate a usar la versión que no usa operador de comparación. Es más legible.

```
Dame un número: 100
6 es un número perfecto
28 es un número perfecto
None
```

Mira la última línea, que muestra el contenido de *resultado*. Recuerda que Python usa *None* para indicar un valor nulo o la ausencia de valor, y una función que «no devuelve nada» devuelve la «ausencia de valor», ¿no?

Cambiamos de tercio. Supón que mantenemos dos listas con igual número de elementos. Una de ellas, llamada *alumnos*, contiene una serie de nombres y la otra, llamada *notas*, una serie de números flotantes entre 0.0 y 10.0. En *notas* guardamos la calificación obtenida por los alumnos cuyos nombres están en *alumnos*: la nota *notas*[*i*] corresponde al estudiante *alumnos*[*i*]. Una posible configuración de las listas sería ésta:

```
1 alumnos = ['Ana Pi', 'Pau López', 'Luis Sol', 'Mar Vega', 'Paz Mir']
2 notas   = [10,       5.5,         2.0,        8.5,        7.0]
```

De acuerdo con ella, el alumno Pau López, por ejemplo, fue calificado con un 5.5.

Nos piden diseñar un procedimiento que recibe como datos las dos listas y una cadena con el nombre de un estudiante. Si el estudiante pertenece a la clase, el procedimiento imprimirá su nombre y nota en pantalla. Si no es un alumno incluido en la lista, se imprimirá un mensaje que lo advierta.

Valor de retorno o pantalla

Te hemos mostrado de momento que es posible imprimir información directamente por pantalla desde una función (o procedimiento). Ojo: sólo lo hacemos cuando el propósito de la función es mostrar esa información. Muchos aprendices que no han comprendido bien el significado de la sentencia **return**, la sustituyen por una sentencia **print**. Mal. Cuando te piden que diseñes una función que *devuelva* un valor, te piden que lo haga con la sentencia **return**, que es la única forma válida (que conoces) de devolver un valor. Mostrar algo por pantalla no es devolver ese algo. Cuando quieran que muestres algo por pantalla, te lo dirán explícitamente.

Supón que te piden que diseñes una función que reciba un entero y devuelva su última cifra. Te piden esto:

```
def ultima_cifra(n):
    return n % 10
```

No te piden esto otro:

```
def ultima_cifra(n):
    print n % 10
```

Fíjate en que la segunda definición hace que la función no pueda usarse en expresiones como esta:

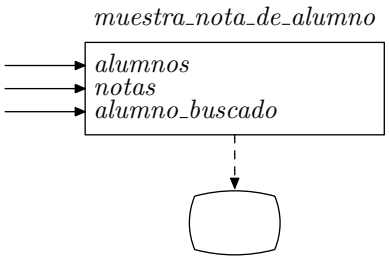
```
a = ultima_cifra(10293) + 1
```

Como *ultima_cifra* no *devuelve* nada, ¿qué valor se está sumando a 1 y guardando en a?

¡Ah! Aún se puede hace peor. Hay quien define la función así:

```
def ultima_cifra():
    n = int(raw_input('Dame un número: '))
    print n % 10
```

No sólo demuestra no entender qué es el valor de retorno; además, demuestra que no tiene ni idea de lo que es el paso de parámetros. Evita dar esa impresión: lee bien lo que se pide y usa parámetros y valor de retorno a menos que se te diga explícitamente lo contrario. Lo normal es que la mayor parte de las funciones produzcan datos (devueltos con **return**) a partir de otros datos (obtenidos con parámetros) y que el programa principal o funciones muy específicas lean de teclado y muestren por pantalla.



Aquí tienes una primera versión:

clase_3.py — clase.py

```
def muestra_nota_de_alumno(alumnos, notas, alumno_buscado):
    encontrado = False
    for i in range(len(alumnos)):
        if alumnos[i] == alumno_buscado:
            print alumno_buscado, nota[i]
            encontrado = True
    if not encontrado:
        print 'El alumno %s no pertenece al grupo' % alumno_buscado
```

Lo podemos hacer más eficientemente: cuando hemos encontrado al alumno e impreso el correspondiente mensaje, no tiene sentido seguir iterando:

clase_4.py — clase.py

```
def muestra_nota_de_alumno(alumnos, notas, alumno_buscado):
    encontrado = False
    for i in range(len(alumnos)):
        if alumnos[i] == alumno_buscado:
            print alumno_buscado, nota[i]
            encontrado = True
            break
    if not encontrado:
        print 'El alumno %s no pertenece al grupo' % alumno_buscado
```

Esta otra versión es aún más breve[3]:

clase.py

```
def muestra_nota_de_alumno(alumnos, notas, alumno_buscado):
    for i in range(len(alumnos)):
        if alumnos[i] == alumno_buscado:
            print alumno_buscado, nota[i]
            return
    print 'El alumno %s no pertenece al grupo' % alumno_buscado
```

Los procedimientos aceptan el uso de la sentencia **return** aunque, eso sí, sin expresión alguna a continuación (recuerda que los procedimientos no devuelven valor alguno). ¿Qué hace esa sentencia? Aborta inmediatamente la ejecución de la llamada a la función. Es, en cierto modo, similar a una sentencia **break** en un bucle, pero asociada a la ejecución de una función.

EJERCICIOS

▶ **307** En el problema de los alumnos y las notas, se pide:

a) Diseñar un *procedimiento* que reciba las dos listas y muestre por pantalla el nombre de todos los estudiantes que aprobaron el examen.

b) Diseñar una *función* que reciba la lista de notas y devuelva el número de aprobados.

c) Diseñar un *procedimiento* que reciba las dos listas y muestre por pantalla el nombre de todos los estudiantes que obtuvieron la máxima nota.

d) Diseñar un *procedimiento* que reciba las dos listas y muestre por pantalla el nombre de todos los estudiantes cuya calificación es igual o superior a la calificación media.

e) Diseñar una *función* que reciba las dos listas y un nombre (una cadena); si el nombre está en la lista de estudiantes, devolverá su nota, si no, devolverá *None*.

▶ **308** Tenemos los tiempos de cada ciclista y etapa participantes en la última vuelta ciclista local. La lista *ciclistas* contiene una serie de nombres. La matriz *tiempos* tiene una fila por cada ciclista, en el mismo orden con que aparecen en *ciclistas*. Cada fila tiene el tiempo en segundos (un valor flotante) invertido en cada una de las 5 etapas de la carrera. ¿Complicado? Este ejemplo te ayudará: te mostramos a continuación un ejemplo de lista *ciclistas* y de matriz *tiempos* para 3 corredores.

```
ciclistas = ['Pere Porcar', 'Joan Beltran', 'Lledó Fabra']
tiempo = [[10092.0, 12473.1, 13732.3, 10232.1, 10332.3],
          [11726.2, 11161.2, 12272.1, 11292.0, 12534.0],
          [10193.4, 10292.1, 11712.9, 10133.4, 11632.0]]
```

En el ejemplo, el ciclista Joan Beltran invirtió 11161.2 segundos en la segunda etapa.
Se pide:

- Una función que reciba la lista y la matriz y devuelva el ganador de la vuelta (aquel cuya suma de tiempos en las 5 etapas es mínima).
- Una función que reciba la lista, la matriz y un número de etapa y devuelva el nombre del ganador de la etapa.
- Un procedimiento que reciba la lista, la matriz y muestre por pantalla el ganador de cada una de las etapas.

[3] . . . aunque puede disgustar a los puristas de la programación estructurada. Según estos, sólo debe haber un punto de salida de la función: el final de su cuerpo. Salir directamente desde un bucle les parece que dificulta la comprensión del programa.

6.2.5. Funciones que devuelven varios valores mediante una lista

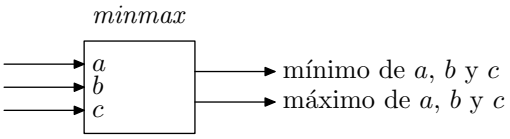
En principio una función puede devolver un solo valor con la sentencia **return**. Pero sabemos que una lista es un objeto que contiene una secuencia de valores. Si devolvemos una lista podemos, pues, devolver varios valores.

Por ejemplo, una función puede devolver al mismo tiempo el mínimo y el máximo de 3 números:

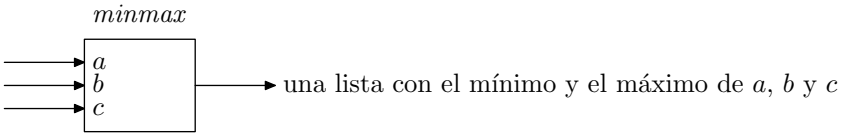
minmax_6.py minmax.py

```
def minmax(a, b, c):
    if a < b:
        if a < c:
            min = a
        else:
            min = c
    else:
        if b < c:
            min = b
        else:
            min = c
    if a > b:
        if a > c:
            max = a
        else:
            max = c
    else:
        if b > c:
            max = b
        else:
            max = c
    return [min, max]
```

Podemos representar a la función con este diagrama:



aunque quizá sea más apropiado este otro:



¿Cómo podríamos llamar a esa función? Una posibilidad es ésta:

minmax_7.py minmax.py

```
⋮
a = minmax(10, 2, 5)
print 'El mínimo es', a[0]
print 'El máximo es', a[1]
```

Y ésta es otra:

minmax_8.py minmax.py

```
⋮
[minimo, maximo] = minmax(10, 2, 5)
print 'El mínimo es', minimo
print 'El máximo es', maximo
```

En este segundo caso hemos asignado una lista a otra. ¿Qué significa eso para Python? Pues que cada elemento de la lista a la derecha del igual debe asignarse a cada variable de la lista a la izquierda del igual.

EJERCICIOS

▶ **309** ¿Qué aparecerá por pantalla al ejecutar este programa?

```
a = 1
b = 2
[a, b] = [b, a]
print a, b
```

▶ **310** Diseña una función que reciba una lista de enteros y devuelva los números mínimo y máximo de la lista simultáneamente.

▶ **311** Diseña una función que reciba los tres coeficientes de una ecuación de segundo grado de la forma $ax^2 + bx + c = 0$ y devuelva una lista con sus soluciones reales. Si la ecuación sólo tiene una solución real, devuelve una lista con dos copias de la misma. Si no tiene solución real alguna o si tiene infinitas soluciones devuelve una lista con dos copias del valor *None*.

▶ **312** Diseña una función que reciba una lista de palabras (cadenas) y devuelva, simultáneamente, la primera y la última palabras según el orden alfabético.

Inicialización múltiple

Ahora que sabes que es posible asignar valores a varias variables simultáneamente, puedes simplificar algunos programas que empiezan con la inicialización de varias variables. Por ejemplo, esta serie de asignaciones:

```
a = 1
b = 2
c = 3
```

puede reescribirse así:

```
[a, b, c] = [1, 2, 3]
```

Mmmm. Aún podemos escribirlo más brevemente:

```
a, b, c = 1, 2, 3
```

¿Por qué no hacen falta los corchetes? Porque en este caso estamos usando una estructura ligeramente diferente: una *tupla*. Una tupla es una lista inmutable y no necesita ir encerrada entre corchetes.

Así pues, el intercambio del valor de dos variables puede escribirse así:

```
a, b = b, a
```

Cómodo, ¿no crees?

6.3. Un ejemplo: Memorión

Ya es hora de hacer algo interesante con lo que hemos aprendido. Vamos a construir un sencillo juego solitario, Memorión, con el que aprenderemos, entre otras cosas, a manejar el ratón desde PythonG. Memorión se juega sobre un tablero de 4 filas y 6 columnas. Cada celda del tablero contiene un símbolo (una letra), pero no es visible porque está tapada por una baldosa. De cada símbolo hay dos ejemplares (dos «a», dos «b», etc.) y hemos de emparejarlos. Una jugada consiste en levantar dos baldosas para ver las letras que hay bajo ellas. Primero se levanta una y después otra. Si las letras que ocultan son iguales, las baldosas se retiran del tablero, pues hemos conseguido un emparejamiento. Si las letras son diferentes, hemos de volver a taparlas. El objetivo es emparejar todas las letras en el menor número de jugadas.

Esta figura te muestra una partida de Memorión ya empezada: