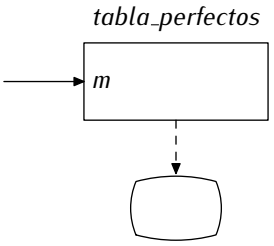
6.2.4. Procedimientos: funciones sin devolución de valor

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:

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

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.

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:

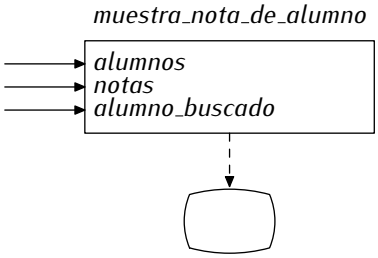
```
a = ultima_cifra(10293) + 1
```

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:
```

```
8       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

```
1 def muestra_nota_de_alumno(alumnos, notas, alumno_buscado):
2     encontrado = False
3     for i in range(len(alumnos)):
4         if alumnos[i] == alumno_buscado:
5             print alumno_buscado, nota[i]
6             encontrado = True
7             break
8     if not encontrado:
9         print 'El alumno %s no pertenece al grupo' % alumno_buscado
```

Esta otra versión es aún más breve[3]:

clase.py — clase.py

```
1 def muestra_nota_de_alumno(alumnos, notas, alumno_buscado):
2     for i in range(len(alumnos)):
3         if alumnos[i] == alumno_buscado:
4             print alumno_buscado, nota[i]
5             return
6     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.

[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.

```
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.