6.2.3. Definición y uso de funciones sin parámetros

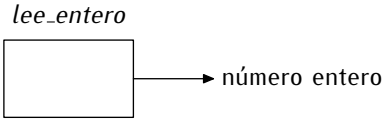
Vamos a considerar ahora cómo definir e invocar funciones sin parámetros. En realidad hay poco que decir: lo único que debes tener presente es que es obligatorio poner paréntesis a continuación del identificador, tanto al definir la función como al invocarla.

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

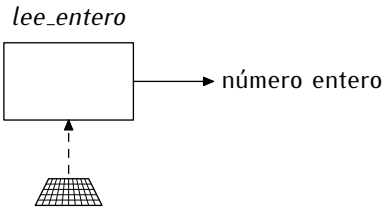
lee_entero.py

```
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.

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.

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

lee_positivo_2.py — lee_positivo.py

```
1 def lee_entero_positivo():
2     numero = int(raw_input())
3     while numero < 0:
4         numero = int(raw_input())
5     return numero
6
7 a = lee_entero_positivo()
```

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

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()
```

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>
```

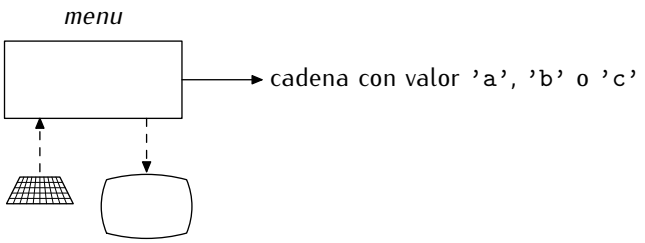
Como puedes ver, el último resultado no es la impresión del mensaje «`Hola`», sino otro encerrado entre símbolos de menor y mayor. Estamos llamando incorrectamente a la función: *saluda*, sin paréntesis, es un «objeto» Python ubicado en la dirección de memoria 8160854 en hexadecimal (número que puede 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 295) te proporcionará más información.

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

```
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
```



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

```
def menu():
    opcion = ''
    while len(opcion) != 1 or opcion not in 'abc':
        print 'Cajero automático.'
        print 'a) Ingresar dinero.'
        print 'b) Sacar dinero.'
        print 'c) Consultar saldo.'
        opcion = raw_input('Escoja una opción: ')
        if len(opcion) != 1 or opcion not in 'abc':
            print 'Sólo puede escoger las letras a, b o c. Inténtelo de nuevo.'
    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:

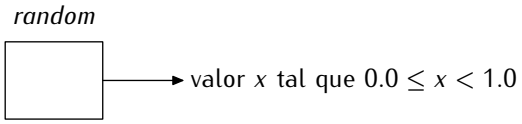
menu_generico(['Saludar', 'Despedirse', 'Salir'])

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.