Funciones en Python

Así como el lenguaje tiene funciones predefinidas por defecto o agrupadas en módulos según distintas especialidades, como las funciones matemáticas en math, en un programa se pueden definir funciones para descomponer el problema complejo que éste debe resolver en subproblemas más simples que permitan sistematizar la resolución mediante un método denominado “dividir y conquistar” o, también, “refinamientos sucesivos”.

En el marco de la resolución de un problema complejo, las funciones también son especialmente útiles cuando un mismo subproblema debe resolverse en distintos escenarios de un mismo programa, para no tener que repetir el mismo código en distintas partes de éste.

```
1  '''Aproximación del valor de pi'''
2  # DEFINICIÓN DE FUNCIONES
3  def fact(n):
4      '''Factorial de un entero no negativo n
5      0!=1
6      1!=1
7      n!=1*2*3*...*(n-1)*n'''
8
9      #PRÓLOGO
10     # Establecimiento de valores iniciales de variables auxiliares o de resultados que se calcularán
11     if n==0 or n==1: return 1 # termina la función devolviendo el valor 1
12     else:
13         P = n # para desarrollar P = n*(n-1)*...*3*2
14         n -= 1
15
16     #DESARROLLO
17         while n>1: # mientras n no llegue a 1
18             P *= n # multiplicar fact por n
19             n -= 1 # restar 1 a n
20
21     #EPÍLOGO
22         return P # termina la función devolviendo el valor del producto P
23
24 def Ramanujan():
25     '''Aproximación del valor de pi mediante la fórmula de Ramanujan:
26     1/pi=2*2**0.5/9801*suma(((4*n)!*(26390*n + 1103))/((n!)**4 * 396**(4*n)))
27     desde n=0 hasta infinito'''
28
29     # PRÓLOGO
30     n = 0
31     sumando = 1103 # sumando para n=0
32     suma = sumando
33     límite = 1.0e-15 # valor límite para calcular términos de la serie
34
35     # DESARROLLO
36     while sumando>límite:
37         n += 1
38         sumando = (fact(4*n)*(26390*n + 1103))/(fact(n)**4 * 396**(4*n))
39         suma += sumando
40         # print(sumando) # impresión de control durante el desarrollo
41
42     # EPÍLOGO
43     return 2.0 * 2.0**0.5 / 9801.0 * suma
44
45 # PROGRAMA
46 print(f'Aproximación de pi por Ramanujan: {1.0/Ramanujan()}')
47 from math import pi
48 print(f'Aproximación de pi por math:      {pi}')
49 input('Pulse Enter para terminar...')
```

Como puede apreciarse en el programa previo, la función fact() se invoca dos veces en la línea 38 para resolver una aproximación de 1/pi mediante otra función Ramanujan(), que a su vez se invoca en la línea 46.

La palabra clave def introduce la definición de una función. Debe ir seguida del nombre de la función y de la lista, que puede ser vacía, de parámetros **formales** entre paréntesis. Las sentencias que forman el cuerpo de la función comienzan en la línea siguiente y deben ir sangradas.

Los parámetros que se utilizan en la definición de funciones se denominan formales porque representan cualesquiera argumentos concretos con los que puedan invocarse (valores literales, variables o expresiones). Los parámetros que se utilizan para invocar una función se denominan **reales**. Por ejemplo, en el programa

anterior, los parámetros reales con los que se invoca la función fact() en la línea 38 son la expresión 4*n y la variable n, respectivamente. El establecimiento del valor del sumando o término 0 de la serie involucrada en la fórmula de Ramanujan para aproximar 1/pi en la línea 31 también podría haberse calculado mediante la expresión

```
sumando = (fact(4*0)*(26390*0 + 1103))/(fact(0)**4 * 396**(4*0))
```

pero para obviar operaciones con resultado conocido y hacer más eficiente el programa evitando cálculos innecesarios, se asigna el valor simplificado 1103…

Parámetros especiales

Por defecto, los argumentos pueden pasarse a una función Python por posición o explícitamente por palabra clave. Por razones de legibilidad y rendimiento, tiene sentido restringir la forma en que se pueden pasar los argumentos, de forma que un desarrollador sólo tenga que mirar la definición de la función para determinar si los elementos se pasan por posición, por posición o palabra clave, o por palabra clave.

Una definición de función puede tener este aspecto

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

en la que pos1 y pos2 son parámetros posicionales, pos_or_kwd puede pasarse tanto posicionalmente como por palabra clave, y kwd1 y kwd2 deben pasarse exclusivamente por palabra clave.

Los símbolos / y * son opcionales, y si se usan indican el tipo de parámetro por el que se pueden pasar los argumentos a la función: sólo posicional, posicional-o-palabra-clave o sólo-palabra-clave. Los parámetros de palabra clave también se denominan parámetros con nombre.

Considere los siguientes ejemplos de definición de funciones prestando especial atención a los marcadores / y *:

```
def standard_arg(arg):
    print(arg)

def pos_only_arg(arg, /):
    print(arg)

def kwd_only_arg(*, arg):
    print(arg)

def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)
```

La primera definición de función, standard_arg, la forma más familiar, no impone restricciones a la convención de llamada y los argumentos pueden pasarse por posición o palabra clave:

```
standard_arg(2) # devuelve 2

standard_arg(arg=2) # devuelve 2
```

La segunda función pos_only_arg está restringida para usar sólo parámetros posicionales ya que hay un / en la definición de la función:

```
pos_only_arg(1) # devuelve 1

pos_only_arg(arg=1) # produce error por pasar argumento por nombre
```

La tercera función kwd_only_args sólo permite argumentos de palabra clave como se indica con un * en la definición de la función:

```
kwd_only_arg(3) # produce error por pasar un argumento posicional

kwd_only_arg(arg=3) # devuelve 3
```

Y la última utiliza las tres convenciones de llamada en la misma definición de función:

```
combined_example(1, 2, 3) # produce error por pasarse 3 argumentos
# posicionales cuando sólo requiere 2

combined_example(1, 2, kwd_only=3) # devuelve 1 2 3

combined_example(1, standard=2, kwd_only=3) # devuelve 1 2 3

combined_example(pos_only=1, standard=2, kwd_only=3) # produce error
# por pasarse un argumento posicional con nombre pos_only
```

Como orientación:

- Utilice sólo posicionales si desea que el nombre de los parámetros no esté disponible para el usuario. Esto es útil cuando los nombres de los parámetros no tienen un significado real, si quiere imponer el orden de los argumentos cuando se llama a la función o si necesita tomar algunos parámetros posicionales y palabras clave arbitrarias.
- Utilice sólo palabras clave cuando los nombres tengan significado y la definición de la función sea más comprensible al ser explícita con los nombres o si desea evitar que los usuarios confíen en la posición del argumento que se pasa.