Python - Funciones

¿Qué es una función?

En matemática, es una transformación de un número (o números) en otro. Recuerden las funciones trigonométricas, o la función cuadrática. En general se definen como $f(x)=x^2$, donde "f" es el nombre de la función, x es la variable, y x^2 lo que hace la función. Pero esto no nos sirve de mucho para aprender a programar.

En la vida cotidiana, por ejemplo, mi microondas tiene la función de descongelar. Un función suele ser un modo de funcionamiento de un aparato. Tampoco de gran ayuda ¿no?

En programación (ahora si a lo nuestro), es un trozo de código con un nombre que podemos ejecutar desde cualquier lugar de nuestro programa. Una función puede o no tener parámetros de entrada y puede o no devolver uno (o más) valores de salida.

¿Cómo se define? ¿Cómo se usan?

Veamos algunos ejemplos simples. En Python se definen de la siguiente manera:

```
def Suma1(a,b):
   resultado = a + b
   return resultado
```

La primer línea, define la función, **def** es la instrucción de Python para iniciar la definición de una función. **Suma1** es el nombre de la función, y **(a,b)** son los parámetros de la función (valores de entrada).

La segunda línea calcula el resultado. No se olviden que todas las líneas de un bloque de programa tiene que ir indentadas con la misma cantidad de espacio, excepto que dentro de un bloque halla otro que va indentado nuevamente.

La tercer línea, devuelve el valor calculado. **Return** es la instrucción para terminar una función, devolviendo un valor, y **resultado** es el valor a devolver.

Y para ejecutar, o como se suele decir, llamar a la función:

```
>>> Suma1(3,4)
7
```

También podríamos haber hecho la función un más corta:

```
def Suma2(a,b):
   return (a + b)
```

Y el resultado es el mismo.

```
>>> Suma2(3,4)
7
```

Pero no se olviden, el nombre de la función tiene que ser lo suficientemente descriptiva de lo que hace la función. No usen el primer nombre que se les ocurra, un nombre ambiguo o un nombre que tiene otro significado, y , por supuesto, el nombre de una función no puede ser una palabra reservada de Python.

Por ejemplo:

```
def Multiplicacion(a,b):
   return (a + b)
```

Probemos, y miremos el resultado.

```
>>> Multiplicacion(2,2)
4
>>> Multiplicacion(3,4)
7
```

En la primer llamada, parece funcionar bien. Pero en la segunda devuelve cualquier cosa. El problema no es la función que está perfectamente definida, el problema es el nombre de la función, nosotros esperamos otro comportamiento.

NOTA: Todas las variables usadas dentro de una función, solo existen dentro de la función, una vez que salimos de la función, esas variables dejan de existir. Se dice que las variables son locales en la función.

En el ejemplo anterior.

```
>>> Suma1(3,4)
7
>>> print(resultado)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    print(resultado)
NameError: name 'resultado' is not defined
```

La instrucción print() devuelve un error porque la variable resultado no existe fuera de la función.

¿Para qué sirve una función?

Ya sabemos cómo se definen y usan, pero esto tampoco nos sirve de mucho. La mayoría de los tutoriales o cursos nos explican como se hacen y usan, pero falta lo más importante, ¿para qué las usamos?

Veamos los usos más frecuentes de funciones y cómo podemos aprovecharlas en nuestros programas. Algunos usos pueden parecer iguales o similares, y es probable que tengan razón, pero la idea es que vean cómo el uso de funciones les puede simplificar el programa.

NOTA: Como Python es un lenguaje interprete (no compilado) ejecuta las instrucciones a medida que le llegan, una tras otra. Por eso, las funciones tienen que estar definidas antes que las llame el programa.

1) Vamos por partes (dijo Jack).

Pueden separar un programa en partes para simplificar el código, y/o para repartir la tarea de programar entre varias personas. Esto permite dos cosas, 1 concentrarse en un aspecto del problema sin tener en cuenta el resto, y 2 que cada programador pueda trabajar en paralelo y/o se especialice en una parte del problema.

Por ejemplo tenemos que hacer un programa para jugar al Ajedrez, el programa podría quedar algo así:

```
#Programa de Ajedrez
Tablero = []
Inicializacion(Tablero)
while (True):
   TurnoBlancas(Tablero)
   Mostrar(Tablero)
   ComprobarJaqueMate(Tablero)
   TurnoNegras(Tablero)
   Mostrar(Tablero)
   ComprobarJaqueMate(Tablero)
# Fin del programa
```

Simple ¿no? Claro faltan definir las funciones (que son las complejas). Pero estamos viendo como el uso de funciones simplifica el programa, no cómo hacemos dichas funciones. Por cierto pueden usar funciones dentro de las funciones para simplificar.

Ahora también pueden dividirse el programa y un programador resuelve la función `ComprobarJaqueMate()`, otro la función `TurnoBlancas()` y `TurnoNegras()`, etc.

2) Divide y vencerás (atribuido a Filipo II de Macedonia).

Otra posibilidad es dividir un problema complejo en problemas más simples de solucionar, y luego combinar dichas soluciones para resolver el problema original.

Por ejemplo, veamos como calcular la superficie de una cara de una arandela (no sé para qué queremos la superficie de una arandela, pero nos sirve de ejemplo). Existe una fórmula para calcularla, dudo que se la acuerden de memoria. Más simple es calcular la superficie de la circunferencia exterior y restarle la superficie del agujero.

```
#Importamos la constante Pi que la usaremos para los cálculos
from math import pi

def SuperficieCircunf (Diametro):
   """Calcula la superficie de una Circunferencia, dado su diámetro"""
   return (pi * Diametro**2 / 4)

#Programa
D_Mayor = float(input("Ingrese el diámetro mayor:"))
D_Menor = float(input("Ingrese el diámetro menor:"))
Superficie = SuperficieCircunf (D_Mayor) - SuperficieCircunf (D_Menor)
print("Superficie:",Superficie)
```

En este caso dividimos el cálculo en una resta de dos cálculos más sencillos, además aprovechamos y los dos cálculos se resuelven con la misma función.

Dividir el problema en problemitas que sean fácil de resolver permite no abrumarse con el problema. Pero no subdividan un problema que ya saben resolver, o que tengan un operador que lo resuelva. La función Suma1() del principio es totalmente inútil, el operador "+" ya lo resuelve.

3) Cuando una pate del código tenemos que repetirla, y repetirla, y repetirla, y repetirla, y repetirla...

En algunos casos vemos que, para resolver un problema, hay una parte del código que debemos repetir varias veces. En estos casos podemos, como se dice en matemática, sacar factor común, y convertir ese trozo de código en una función, al llamarla con distintos valores de los parámetros nos sirva para cada caso.

Por Ejemplo, veamos el programa de la caja del supermercado. Tenemos que ir sumando el precio de cada producto, hasta aquí todo fácil. Pero resulta que hay productos que tienen un descuento determinado, descuentos sobre el total de la compra, descuentos si se compran 2 productos iguales, descuentos en la segunda unidad, etc. Podemos en realizar una función que tiene como parámetros el precio del producto y el descuento, y nos devuelve el valor final.

```
def CalculoDescuento(Precio,Descuento):
   """Calcula el precio final,
   dado el precio del producto y el porcentaje de descuento"""
   return (Precio - Precio * Descuento/100)
```

La función es simple, y podemos utilizarla donde necesitemos, al calcular el precio de un producto, al calcular el total de la compra, etc.

Esto permite reutilizar una parte del código dentro de un proyecto, y también entre proyectos. Podemos reunir las funciones en un archivo e importarlas en todos los proyectos que las necesitemos. Pero para esto debemos tener en cuenta que las funciones sean robustas.

4) Resolver un problema, que requiere resolver un problema, que requiere resolver un problema, que requiere resolver un problema, que requiere resolver un problema, que requiere.....

En general encontramos este tipo de problemas en matemática e ingeniería, como métodos numéricos para resolver un problema (ver calculo numérico). Hay dos clases de problemas que se pueden solucionar mediante repetir la función una y otra vez.

Los problemas **Iterativos**: La función calcula una solución aproximada a un problema. Luego se usa esa solución como dato para la misma función, que calcula una solución **más** aproximada a un problema. Luego se usa esa solución como dato para la misma función, que calcula.... Con cada **iteración** se obtiene una solución más acotada (más precisa) hasta que se obtenga la solución con la precisión requerida.

Los problemas **Recursivos:** La función sabe resolver algunos casos sencillos. Si el caso es más complejo, la función sabe como operar con los datos para reducir su complejidad y **se llama a sí misma** con estos nuevos datos hasta llegar a un caso sencillo que la función sabe resolver. Esto permite que la función sea sencilla de escribir, y con las sucesivas llamadas a sí misma obtener el resultado.

La diferencia entre ambos métodos puede ser sutil, y si les parece lo mismo, no se preocupen, muchas veces se confunden.

Veamos un ejemplo de cada uno.

Ejemplo típico de problema **iterativo**: El método de bisección se usa para ubicar una raíz de una función dentro de un intervalo. En criollo, encontrar cuando la función es igual a 0. Vamos a definir una función f(x) que es la función de la que queremos buscar la raíz, y la función Bisección que busca la raíz de forma iterativa, en cada iteración reduce el intervalo de búsqueda a la mitad.

https://es.wikipedia.org/wiki/M%C3%A9todo_de_bisecci%C3%B3n

```
def f(x):
   """Función a calcular la raíz"""
   return (x**2 - 2*x -2)

def Biseccion(a,b,error):
   """Busca una raíz de f(x) dentro del intervalo a,b.
   Itera hasta que |b-a|<error"""
   while abs(b-a) > error: #Inicio de la iteración
      medio = (a + b)/2    #calcula el punto medio
      if f(a)>0 and f(medio)<0: #Si la raíz esta entre a y medio
         b=medio           #Actualizo el intervalo
      if f(a)<0 and f(medio)>0: #Si la raíz esta entre a y medio
         b=medio           #Actualizo el intervalo
      if f(medio)>0 and f(b)<0: #Si la raíz esta entre medio y b
         a=medio           #Actualizo el intervalo
      if f(medio)<0 and f(b)>0: #Si la raíz esta entre medio y b
         a=medio           #Actualizo el intervalo
   #Fin de la iteracion
   return medio
```

Y un ejemplo de su uso.

```
>>> Biseccion (-1,0,0.0001)
-0.73199462890625
```

Hay otra raíz dentro entre 2 y 3.¿Se animan a buscarla?

Ejemplo típico de problema **recursivo**: La sucesión de Fibonacci. Los primeros dos términos son 1 y 1, y cada término posterior es la suma de los dos anteriores. El tercer término es 1+1=2, el cuarto es 3, el quinto es 5, etc. El problema es calcular el término N de la sucesión (dado N como dato).

https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci

```
def Fibonacci(N):
   """Calcula el termino N de la sucesión de Fibonacci
   1, 1, 2, 3, 5, 8..."""
   if N == 1:
      return 1
   elif N == 2:
      return 1
   else:
      return (Fibonacci(N-2) + Fibonacci(N-1))
```

Y algunos ejemplos de llamadas a la función.

```
>>> Fibonacci(20)
6765
>>> Fibonacci(30)
832040
```

Los métodos Iterativos y recursivos pueden ser simples de programar, pero, en general, pueden ser lentos. El algunos casos recurrir a recursión o iteración es la única solución posible, y se buscan métodos para agilizar el proceso. En general, se aumenta la velocidad a cambio de recursos de la computadora, por ejemplo memoria RAM. Igualmente uno disfruta de estos métodos cuando vemos cualquier pelicula de animación 3D, los algoritmos para generar las imágenes (rendering) en general son recursivos.

Para comprobar la diferencia de velocidad, prueben hacer:

```
Fibonacci(35)
```

Les dejo una versión de Fibonacci más eficiente en tiempo. Vuelvan a probar lo con la nueva versión y vean la diferencia de tiempo. Si no entienden exactamente cómo funciona el programa no se preocupen (a mi me costó un ratito que me quede prolijo). Lo que hace es guardar cálculos intermedios en la variable `Fibonacci_cache`.

```
def Fibonacci(N):
   """Calcula el termino N de la sucesión de Fibonacci
   1, 1, 2, 3, 5, 8...(usando memorización)"""
   Fibonacci_cache={}
   def Calculo(N):
      nonlocal Fibonacci_cache
      if N in Fibonacci_cache:
         return Fibonacci_cache[N]
      elif N == 1:
         return 1
      elif N == 2:
         return 1
      else:
         nuevo = (Calculo(N-2) + Calculo(N-1))
         Fibonacci_cache[N] = nuevo
         return nuevo
   return Calculo(N)
```

4) Funciones dentro de una manzana.

Me voy a adelantarme un poco a la programación orientada a objetos. En la programación Orientada a Objetos, un objeto reúne los datos que guarda y las funciones para transformar dichos datos. Por lo tanto al definir una clase de objetos, también tenemos que definir las funciones (normalmente conocidas como métodos). Estas funciones se declaran exactamente igual, pero se definen dentro de la definición de la clase de objetos. No voy a poner un ejemplo ahora de esto, lo dejo para más adelante cuando veamos objetos.

Nota: funciones y tipo de datos en Python.

Cuando llamamos a una función , normalmente le pasamos una variable del programa como parámetro de la función. Por ejemplo vean el punto 2) en el programa tenemos la variable `D_Mayor`, pero dentro de la función la variable se llama `Diametro`. Si dentro de la función modificamos el valor de `Diametro` (no es el caso en esta función), al volver al programa la variable `D_Mayor` mantiene el valor original (no se modifica). Esto es lo esperado, y en valido para datos atómicos (que no pueden desmenuzarse en datos individuales) como enteros, decimales, textos, etc.

Pero no es válido para cualquier tipo de lista u objetos. Si pasamos una lista como parámetro de una función y la función modifica la lista, al salir de la función la lista queda modificada. Esto puede ser bueno o malo según como se use, hay que tenerlo en cuenta al programar. Para evitar esto, el programa o la función tiene que hacer una copia de la lista antes de modificarla y trabajar con la copia.

Nota: Documentación de funciones en Python.

Dentro de las funciones que estuvimos haciendo habrán observado que la primer línea comienza y termina con """ (tres comillas dobles) y dentro hay una descripción de la función. Estas líneas son la documentación de la función y se muestran cuando una hace help(funcion). Por ejemplo:

```
>>> help(SuperficieCirculo)
Help on function SuperficieCirculo in module __main__:

SuperficieCirculo(Diametro)
    Calcula la superficie de una Circunferencia, dado su diámetro
```