P07 Funciones y recursión

1.- Funciones.

Recordemos que en matemáticas el factorial de un número natural se define como $n! = 1 \cdot 2 \cdot \ldots \cdot n$ (el producto de todos los números enteros desde 1 a n. Por ejemplo $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. Está claro que el factorial se calcula de forma simple empleando un bucle for.

Imaginemos que necesitamos calcular varios factoriales en un programa, o varias veces el mismo en distintos pasos del código. La solución fácil seria escribirla una vez y usar Copiar-Pegar para repetirlo las veces que sea necesario, obteniendo un código similar al que sigue.

```
 # compute 3!
res = 1
for i in range(1, 4):
    res *= i
print(res)
# compute 5!
res = 1
for i in range(1, 6):
    res *= i
print(res)
```

El principal problema seria que si cometemos un error en el código inicial este código aparecerá en todas los lugares en que lo hayamos copiado. Además el código es bastante más largo de lo que podría ser.

Para evitar reescribir el mismo código una y otra vez aparecen las funciones.

Una **Función** es una sección de código que está separado del resto del programa y que se ejecuta únicamente cuando se la llama. Ya se han empleado varias de ellas como `sqrt()`, `len()` y `print()` . Todas tienen algo en común: Aceptan parámetros (cero, uno o varios) y pueden devolver un valor (aunque pueden también no devolverlo). Por ejemplo, la función `len()` acepta un parámetro y devuelve un valor, mientras que `print()` acepta cualquier número de argumentos, pero no devuelve valor.

Veamos como primer ejemplo como crear una función, llamada `factorial()` , que acepta un único parámetro y devuelve un valor (el factorial de ese número)

```
def factorial(n):
    res = 1
```

```
    for i in range(1, n + 1):
        res *= i
    return res
print(factorial(3))
print(factorial(5))
```

Es necesario explicar varias cosas.

En primer lugar: el código de la función debe colocarse al principio del programa (antes de que se llame en ningún momento a la función)

La primera línea `def factorial(n):` es la descripción de la función, **def** es una palabra clave que indica que comienza una nueva función, la palabra "factorial" es su **identificador** (el nombre de nuestra función)

Después del identificador va una lista de los parámetros que recibe nuestra función (en paréntesis). La lista cosiste en los identificadores que vamos a usar separados por comas. Al final de la línea van dos puntos-

A continuación se encuentra el cuerpo de la función, que en python siempre se delimita empleando la indentación.

En el cuerpo se realizan los cálculos y en la línea `return res` se devuelve el valor, lo que termina la función.

La palabra **return** puede aparecer en cualquier parte de la función, su ejecución devuelve el valor especificado y termina la función. Si no se quiere que devuelva nada se puede emplear igualmente para finalizar la función, aunque no es imprescindible.

Veamos otro ejemplo, la función `max()` que acepta dos números y devuelve su máximo.

```
 def max(a, b):
    if a > b:
        return a
    else:
        return b
print(max(3, 5))
print(max(5, 3))
print(max(int(input()), int(input())))
```

Una vez que la tenemos podemos escribir la función `max3()` que da el máximo de tres números.

```
 def max(a, b):
    if a > b:
        return a
    else:
        return b
def max3(a, b, c):
    return max(max(a, b), c)
print(max3(3, 5, 4))
```

La función incluida en Python puede aceptar cualquier número de parámetros sin tener que decidir cuantos serán a priori. Veamos como se hace

```
def max(*a):
    res = a[0]
    for val in a[1:]:
        if val > res:
            res = val
    return res
print(max(3, 5, 4))
```

Todos los argumentos que se pasan a la función se almacenan en una única lista a, esto se indica usando una * en la definición de la función

2.- Variables locales y globales.

Dentro de una función podemos usar variables que estén definidas fuera de ella (esto no quiere decir que sea buena idea hacerlo)

```
def f():
    print(a)
a = 1
f()
```

En este ejemplo la variable `a` se fija en 1 y después la función `f()` escribe su valor aunque cuando declaramos la función `f` la variable no está inicializada. El motivo es que al llamar a la función `f()` (en la última línea) la variable `a` ya tiene un valor.

Estas variables, que se declaran fuera de la función, pero que están disponibles en la función se llaman **globales.**

Sin embargo si inicializamos variables dentro de la función no se pueden usar fuera de ellas, por ejemplo

```
def f():
    a = 1
f()
print(a)
```

Al ejecutarlo nos devuelve el siguiente error `NameError: name 'a' is not defined`. Estas variables declaradas dentro de una función se llaman **locales,** y dejan de estar disponibles cuando la función finaliza su ejecución.

Lo que puede ser más complicado de entender es lo que ocurre al cambiar el valor de una variable global dentro de una función.

```
def f():
    a = 1
    print(a)
a = 0
f()
print(a)
```

El programa devuelve primero “1” y después “0” pese a que el valor cambió dentro de la función, en el resto del programa se mantiene igual. Esto se hace para “proteger” las variables globales de cambios inadvertidos en la función.

De forma mas técnica el interprete de Python considera una variable local en una función. Si en el código hay alguna instrucción que modifique el valor de la

variable entonces no se puede usar antes de su inicialización. Veámoslo con un ejemplo.

```
 def f():
    print(a)
    if False:
        a = 0
a = 1
f()
```

Se obtiene el siguiente error `UnboundLocalError: local variable 'a' referenced before assignment`, ya que se usa a antes de una instrucción que podría cambiarla

Si se quiere cambiar una variable global dentro de una función se debe declarar la variable usando la instrucción `global`:

```
def f():
    global a
    a = 1
    print(a)
a = 0
f()
print(a)
```

Compárese con el ejemplo anterior.

El hecho de que pueda hacerse no quiere decir que sea una buena idea en general. Es una práctica que debe evitarse en la medida de lo posible, ya que dificulta en gran medida el reaprovechar el código de unos programas a otros.

Supongamos que queremos calcular el factorial de un número y guardarlo en una variable para usarlo subsiguientemente.

Lo siguiente es un ejemplo **de cómo no debe realizarse:**

```
 def factorial(n):
    global f
    res = 1
    for i in range(2, n + 1):
        res *= i
    f = res
n = int(input())
factorial(n)
print(f)
# seguimos usando f
```

Este ejemplo es de un mal código, es compliado de reutilizar, y si se consiguiese habría que cuidar que no se usara en ningún momento la variable f

Seria mucho mejor emplear el siguiente código

```
# esta function es mas facil de reutilizer
def factorial(n):
    res = 1
    for i in range(2, n + 1):
        res *= i
    return res
# end of piece of code
n = int(input())
f = factorial(n)
print(f)
# doing other stuff with variable f
```

Por último es útil saber que una función puede devolver más de una variable. Vemos como se hace en un ejemplo y como se puede asignar lo que devuelve a varias variable

```
return [a, b]
```

```
n, m = f(a, b)
```

3.- Recursión

En los ejemplos anteriores hemos visto que dentro del código de una función es posible llamar a otra función. También es posible llamarla dentro de los argumentos con los que se invoca una función.

Un comportamiento más complejo pero muy útil es que una función puede llamarse a si misma. Veamos como se hace con un ejemplo.

En temas anteriores hemos calculado la función factorial empleando un bucle, pero es posible emplear dos propiedades, $0!=1$ y $n!=n\cdot(n-1)!$.. Usando esas dos propiedades se puede crear la siguiente función.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))
```

La situación en que una función se llama a si misma se llama **recursión** y una función que trabaja así se denomina **recursiva.**

La programación recursiva es muy potente, pero puede presentar problemas y no siempre ser demasiado efectiva.

El error más común es el llamado **recursión infinita,** que ocurre cuando la cadena de llamadas no termina nunca (hasta que el ordenador se bloquea). Un ejemplo de esto sería el siguiente

```
def f():
   return f()
```

Hay dos motivos habituales para que esto ocurra.

1. Condición de finalización incorrecta (como puede ocurrir en los bucles while). Por ejemplo, si en la función para calcular el factorial olvidamos la línea `if n == 0`, entonces `factorial(0)` llamará a `factorial(-1)`, después a `factorial(-2)`, y no terminará nunca .
2. Llamar a la función con parámetros incorrectos, por ejemplo, si la función `factorial(n)` llama a `factorial(n)` entraremos en un bucle infinito.

Por ultimo se plantea una cuestión, por que desemboca en un bucle infinito?

Ejercicios:

P1 Longitud de un segmento

Dados cuatro números reales que determinan dos puntos del plano (x_1,y_1) y (x_2,y_2) realizar una función que calcula la distancia entre los dos puntos.

Para esto es necesario emplear la función raíz cuadrada. No es una función disponible de forma inmediata en Python. Para usarla la primera línea de nuestro archivo debe ser
import math
y a partir de entonces se podrá calcular como
math.sqrt(x)

P2 Potencia de un número

Dados un número x y un entero n escribir una función potencia(x,n) que devuelva x^n sin emplear el operador **

P3 Longitud de un segmento, de nuevo

Reescribir el problema 1 empleando la función desarrollada en el problema 2

P4 Fibonacci

La sucesión de Fibonacci se define a partir de las siguientes propiedades

$a_0=1$

$a_1=1$

$a_i=a_{i+1}+a_{i-2}$

Escribir una función recursiva que calcule el n-ésimo término de la sucesión