FUNCIONES

Definición:

Una función es un subprograma que recibe cero o más valores de entrada y retorna un único objeto de salida. Es una tarea independiente que puede o no depender de variables externas. Lo ideal es que funcione como "caja negra", es decir, que se la pueda invocar desde cualquier programa cada vez que se la necesite y que realice una función bien específica

Ejemplo:

```
# Ejemplo 1
ni = input("\n Introduzca el número de iteraciones: ");
if floor(ni)~=ni
   disp("Error: El número no es entero");
endif

nv = input("\n Introduzca el número de variables: ");
if floor(nv)~=nv
   disp("Error: El número no es entero");
endif

ne = input("\n Introduzca un número entero: ");
if floor(ne)~=ne
   disp("Error: El número no es entero");
endif
```

En este ejemplo, hay una porción de código que se repite tres veces. Un código mucho más limpio sería:

```
# Ejemplo 1
ni = input("\n Introduzca el número de iteraciones: ");
validar(ni)
nv = input("\n Introduzca el número de variables: ");
validar(nv)
ne = input("\n Introduzca un número entero: ");
validar(ne)
```

Ventajas de utilizar funciones:

1. Solamente se escribe una vez. En el ejemplo anterior, nos ahorramos tener que escribir la misma porción del código una y otra vez. Esto evita errores involuntarios (pero siempre presentes) de transcripción.
2. Si una función es probada y funciona bien, funcionará bien cada vez que se use (siempre y cuando el uso sea el correcto). Esta es una ventaja importante cuando estamos buscando errores presentes en nuestros programas, puesto que evaluamos pedazos cada vez más pequeños lo que facilita la detección de errores.

3. Son portables. Una misma función puede ser útil para distintos casos, distintos programas y distintos programadores.
4. Código más limpio. Al usar funciones reducimos las líneas de código de nuestro programa y por lo tanto se hacen mucho más fáciles de leer y validar su correctitud.
5. Parte un programa en varios subprogramas.

Se pueden observar dos lados de una función: la invocación y la definición.

Invocación: Se refiere a la llamada a la función. Ejemplos en Octave:

```
octave> y=sin(pi);
```

En este caso, `sin` representa el nombre de la función, `pi` es el argumento y la variable `y` el objeto de salida.

Definición: Se refiere al texto de la función con el desarrollo del cuerpo de la misma. En nuestro caso, se trata de archivos .m donde volcamos, en sintaxis correcta, el texto de la función como un conjunto de sentencias Octave válidas:

```
octave> edit


function salida = nombreFuncion (lista_de_parametros)
    …… %sentencias
   salida = ...

end
```

- La palabra FUNCTION es una palabra reservada que puede estar en mayúsculas o minúsculas
- La lista de parámetros debe ir separada por comas (en caso que haya más de un parámetro)
- Si la salida está compuesta por más de un valor, se colocan corchetes y los valores se separan por comas:

  function [$salida_1$,…,$salida_n$] = nombreFuncion (par_1,…, par_m)

- Puede haber funciones sin argumentos y también sin valor de retorno:

```
function hola
   disp('Hola Mundo')
end;
```

Ejercicio:

Realizar la función validar(n) de nuestro ejemplo anterior

Supongamos que queremos realizar una función que retorne la suma de dos escalares.

1)Generamos la función suma:

```
octave> edit
            function   salida = suma(a,b)
                  salida = a+b;
            end;
```

2)Grabar el archivo con el nombre suma.m
3)Invocar a la función desde Octave:

```
octave> y = suma(2,3)
y = 5
```

Generalmente, cuando utilizamos el comando "help nombre" se despliega en pantalla una ayuda que nos permite saber que realiza el comando o función nombre:

```
octave> help sin
```

En este caso, desplegaría una ayuda sobre la función sin (seno), indicando que hace y diferentes modos de uso. Nosotros podemos hacer lo mismo con nuestra función suma. Es más, es recomendable que, cuando definimos una función, coloquemos como encabezado una pequeña descripción de cuál es el objetivo de la función y una sintaxis de uso de la misma:

```
octave> edit
        % retorno = suma(a, b)  donde a y b son valores numéricos
        %
        % Calcula la suma de dos valores
        %*************************************
        function   salida = suma(a, b)
                        salida = a + b;
        end;
```

Notar que el comentario debe estar al comienzo del archivo para que aparezca cuando se realiza help suma

Variables de la función:

¿Qué sucede si por línea de comandos quiero saber qué valor tienen las variables a y b de la función o la variable salida?

```
octave> a
error: 'a' undefined near line 2 column 1
```

El mensaje de error indica que la variable 'a' no es conocida en este contexto. Es decir, las variables de la función sólo tienen validez **dentro de la función misma.** Realicemos un experimento:

```
octave> a=123
a = 123
octave> suma(3,4)
ans = 7
octave> a
a = 123
```
y no 3 que es el valor que toma a dentro de la función!!!

Esto nos indica que las variables poseen diferentes **alcances**

Las variables dentro de una función tienen alcance dentro de la misma función. No interfieren con variables con el mismo nombre declaradas fuera de la misma.
Las variables declaradas en el intérprete (por línea de comando) tienen validez en el intérprete.
Se dice entonces que las variables son **locales** de la función.
Para que la función tenga acceso a variables que no han sido pasadas como argumentos, es necesario declarar a dichas variables como **globales,** tanto en el intérprete como en todas las funciones que deben acceder a su valor
Veamos un ejemplo:

```
octave> x=5
octave> edit suma.m
      % retorno = suma(a, b)  donde a y b son valores numéricos
      %
      % Calcula la suma de dos valores
      %************************************
      function   salida = suma(a, b)
             printf('%d\n', x)
              salida = a + b;
end;
octave> suma(10,3)
error: 'x' undefined near line 2 column 14
error: evaluating argument list element number 1
```

El error surge porque x fue declarada dentro del intérprete Octave pero quiere ser utilizada dentro de la función, donde no tiene **alcance**
Cambiemos lo anterior por:

```
octave> clear x
octave> global x=5
octave> edit suma.m
      % retorno = suma(a, b)  donde a y b son valores numércicos
      %
      % Calcula la suma de dos valores
      %************************************
      function   salida = suma(a, b)
           global x
           printf('%d\n', x)
            salida = a + b;
      end;
octave> suma(10,3)
x = 5
ans = 13
```

Nota: En el segundo ejemplo usamos clear x, para borrar de memoria la variable ya declarada en el primer ejemplo. De esta manera eliminamos la variable local x para luego, en la segunda línea de ejemplo, generar una variable de uso global x.

Luego, dentro de la función, debemos colocar global x para indicar que vamos a usar la variable global x declarada en el programa principal o en el intérprete. Más abajo, ya podemos usar el valor de x sin miedo a que nos aparezca el mensaje de error, dado que dicho valor ya es conocido por la función.

Volviendo a nuestra suma original:

```
octave> edit
        % retorno = suma(a, b)  donde a y b son valores numéricos
        %
        % Calcula la suma de dos valores
        %*************************************
        function   salida = suma(a, b)
                        salida = a + b;
        end;
```

Supongamos ahora que queremos retornar dos valores: la suma de ambos argumentos y la multiplicación de ambos argumentos. Cambiemos la función de la siguiente manera y la grabamos como sumamul.m:

```
octave> edit
        % retorno = sumamul(a, b)  donde a y b son valores numéricos
        %
        % Calcula la suma y la multiplicación de dos valores
        %*************************************
        function   [suma,mul] = sumamul(a, b)
                        suma = a + b;
                        mul = a * b;
        end;
```

Vemos que hay dos valores de retorno: suma, donde queda guardada la suma de los dos argumentos y mul donde quedará la multiplicación. ¿Cómo se usa desde Octave?

```
octave> [s,m]=sumamul(2,3)
 s= 5
 m= 6
```

Es decir, se han generado dos variables, una para cada respuesta que retorna la función.

Captura de errores en los argumentos de entrada y/o salida:

Supongamos que realizamos las siguientes invocaciones:

```
octave> [s,m]=sumamul
octave> [s,m,d]=sumamul(2,3)
octave> [s,m]=sumamul(2)
```

¿Qué tienen en común estas invocaciones?
Que causan algún error

¿Se pueden capturar estos errores?
Sí, siempre que en el texto de la función lo controle

Variables nargin y nargout

Las variables nargin y nargout cuentas la cantidad de variables de entrada a la función y la cantidad de salidas que produce la función.

Veamos como controlamos la cantidad de argumentos de entrada a la función:

```
octave> edit
        % retorno = sumamul(a, b)  donde a y b son valores numéricos
        %
        % Calcula la suma y la multiplicación de dos valores
        %**************************************
        function   [suma,mul] = sumamul(a, b)
          if (nargin==2)
                        suma = a + b;
                        mul = a * b;
          else
             usage('La función necesita dos parámetros')
          end;
        end;
```

Ejercicios:

1) El número combinatorio (n,k) se define como n!/(k!*(n-k)!).
   a. Realizar una función que reciba como parámetros n y k y retorne el combinatorio
   b. Mejorar la función para que, por ejemplo evite la división por cero con un mensaje de error
2) El área de un triángulo de lados a, b y c está dada por la ecuación:

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)},$$

donde s=(a + b + c) / 2. Escribir una función que acepte a, b y c como argumentos de entrada y retorne el área del triángulo como salida.

3) Realice una función fact(n) que retorne el factorial de n. Puede usar la función factorial(n) o la función prod(n). Pero debe verificarse que n sea un número natural > 0. En cualquier otro caso, debe mostrar un mensaje de error
4) Realizar una función que realice la suma de los inversos de los números de 1 a n. La función recibe n como parámetro y retorna la suma. Desde Octave, se pide n al usuario (cuidando que n sea mayor que 0)