FUNCIONES

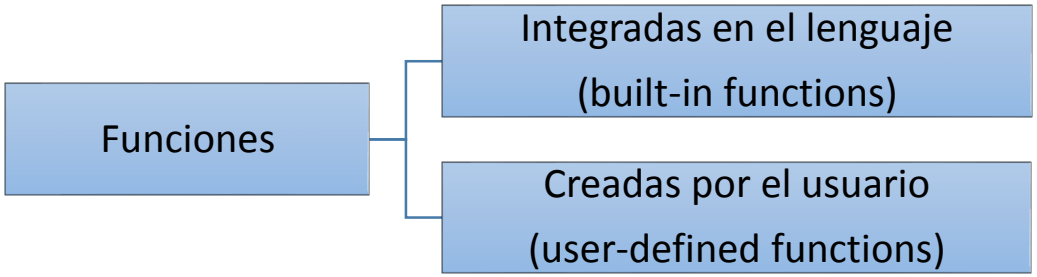
FUNCIONES

En programación, una función se puede ver como un bloque de código *(conjunto de sentencias o instrucciones)* con un nombre asociado, cuya finalidad es la de resolver una cierta tarea.

Entre sus múltiples ventajas podemos destacar las siguientes:

- *Modularización*: permite dividir un programa complejo en partes más simples, haciendo que el programa sea más fácil de leer y facilitando así la programación y el depurado de errores.

- *Reusabilidad*: una función se puede reutilizar tantas veces como sea necesario en el mismo o en diferentes programas sin necesidad de volver a escribir o repetir el código cada vez, haciendo el programa más pequeño y simplificándolo en consecuencia.



- Definición o declaración de una función

En Python, la definición de una función se realiza a través de la cláusula:

```
def nombre_funcion(parámetro_1, parámetro_2, …):      # este es el encabezado de la función
    sentencias_de_la_función                          # este es el cuerpo de la función
```

```
# Ejemplo 01

def hola():
   print("Hola Mundo")
```

- Documentando la función definida: Docstring

La documentación interna del código es importante para clarificar y explicar el comportamiento del mismo de cara sobretodo a revisiones futuras.

La primera sentencia o instrucción de una función normalmente es una cadena de texto que servirá de documentación interna de dicha función. En Python, esta documentación se denomina *docstring.*

La *Docstring* es opcional e irá encerrada entre triples comillas dobles """ … """

```
def nombre_funcion(parámetro_1, parámetro_2, …):      # este es el encabezado de la función
    """ Descripción de la función. """                # esta es la docstring que describe la función
    sentencias_de_la_función                          # este es el cuerpo de la función
```

```
# Ejemplo 02

def hola():
    """Imprime la frase Hola Mundo en pantalla."""
    print("Hola Mundo")
```

- Es una buena práctica habituarse a escribir documentación de las funciones definidas.
- La *docstring* se puede consultar con la instrucción *help(funcion)*. Por ejemplo: *help(hola)*

- Invocación o llamada a una función

La definición de una función no significa que se ejecute. Para su ejecución necesita ser invocada o llamada.

Una función se invoca a través de su nombre y sus argumentos (*entre paréntesis en caso de haberlos).*

```
def nombre_funcion(parámetro_1, parámetro_2, …):      # este es el encabezado de la función
    sentencias_de_la_función                          # este es el cuerpo de la función

nombre_funcion(argumento_1, argumento_2, …)           # aquí se invoca o se llama a la función
```

```
# Ejemplo 03

def hola():
    """Imprime la frase Hola Mundo en pantalla."""
    print("Hola Mundo")

hola()
```

- En este ejemplo se invoca a la función *hola* definida previamente, por lo que se ejecutará y se imprimirá en pantalla el mensaje “Hola Mundo”.
- Es obligatorio poner los paréntesis, tanto en la definición como en la llamada de la función.

- Parámetros de una función (i)

Un parámetro es un valor que necesita la función para su correcta ejecución y que recibirá al ser invocada.

Una función puede recibir ninguno, uno o varios parámetros que irán separados por comas y entre paréntesis.

def nombre_funcion(parámetro_1, parámetro_2, …): *# encabezado de la función con parámetros*
sentencias_de_la_función *# este es el cuerpo de la función*

```
# Ejemplo 04

def imprime_mensaje(mensaje):
    """ Imprime el texto recibido en pantalla. """
    print(mensaje)

imprime_mensaje("Me gusta Python")
```

- La función *imprime_mensaje* se invoca con el argumento de tipo cadena de texto *"Me gusta Python"*.
- El parámetro *mensaje* apuntará al valor pasado como argumento en la invocación de la función.
- La función muestra el contenido de *mensaje* en pantalla a través de la instrucción *print(mensaje)*.

- Parámetros de una función (ii)

Los parámetros actúan como variables locales, es decir, sólo tendrán vigencia dentro de la función a la que pertenecen, pero no fuera de ella.

```
# Ejemplo 05

def nombre_completo(nombre, apellidos):
    """    Imprime el nombre seguido de los apellidos.  """
    nc = nombre + " " + apellidos
    print("Nombre completo:", nc)

nombre_completo("Francisco", “Antequera Muñoz")
print ("Los apellidos son:", apellidos)
```

- La función *nombre_completo* se ejecutaría correctamente y mostraría en pantalla “Francisco Antequera Muñoz”
- La instrucción *print(“Los apellidos son:”, apellidos)* arrojaría el siguiente mensaje de error:

  NameError: name 'apellidos' is not defined

  ya que *apellidos* es una variable local a la función *nombre_completo* y, por tanto, no existe en el cuerpo principal del programa.

- Parámetros de una función (iii)

Se pueden fijar valores por omisión a los parámetros de una función. Estos valores por defecto se usarán si en la invocación a la función no se indican otros.

```
# Ejemplo 06

def pvp(precio, iva = 12):
    """Calcula el precio de venta al publico.
       precio = precio base del articulo sin IVA
       iva = 12% por defecto
    """
    pvp = precio + (precio * iva / 100)
    print ("El precio total es:", pvp)

precio_sin_iva = 100
pvp(precio_sin_iva)
pvp(precio_sin_iva, 21)
```

- La función *pvp* ha sido definida para calcularse con el 12% de IVA por defecto.
- En el primer *print* se omite el argumento *iva* y se imprimiría el mensaje *"El precio con IVA(12%) es: 112".*
- En el segundo print se pasa el valor *21* a la función *pvp* por lo que el parámetro *iva* pasará a valer 21 en lugar de 12 y se imprimirá el mensaje *"El precio con IVA es: 121".*

Importante: los parámetros por omisión se han de definir los últimos.

- Parámetros de una función (iv)

También es posible declarar una función con un número indeterminado o variable de parámetros que se pasarán como una lista arbitraria de argumentos. Esto se hace anteponiendo el símbolo * al nombre del parámetro. El parámetro que tiene el * será del tipo tupla.

```
# Ejemplo 07

def media(*datos):

    """ Devuelve la media aritmetica de los valores recibidos.
        datos es una tupla que contiene la lista de datos para la media
    """
    suma = 0
    for i in datos:
        suma += i
    med = suma / len(datos)
    print("La media es:", med)

media(2, 3, 4, 5, 1, 9, 18)
v1, v2, v3, v4 = 2, 7, 3, 8
media(v1, v2, v3, v4)
```

- La función *media* recibirá un número de argumentos indeterminado.
- En el primer *print* se calculará la media de los valores 2, 3, 4, 5, 1, 9 y 18.
- En el segundo *print* se calculará la media de los valores 2, 7, 3 y 8.

- Devolución de valores (i)

En muchas ocasiones necesitamos que una función devuelva el resultado de sus operaciones. Esto es posible a través de la instrucción *return*.

```
# Ejemplo 08

def saluda():

    """    Devuelve una cadena con un mensaje de saludo. """

    return "Encantado de conocerle";

nombre = "Pepe"
print(saluda(), nombre)
```

- La función *saluda* devuelve la cadena de texto *"Encantado de conocerle."*.
- El *print* del cuerpo principal imprime la cadena de texto "*Encantado de conocerle Pepe*".

Nota: Una función que termine de ejecutarse correctamente sin ninguna sentencia *return*, siempre devolverá el valor *None*.

- Devolución de valores (ii)

Puede haber varias instrucciones *return* en la misma función. La función retornará a donde fue llamada cuando se encuentre el primero de ellos, sin que se ejecuten el resto de sentenicias de la función.

```
# Ejemplo 09

def cadena_larga():

    """ Comprueba si una cadena es mayor de 5 caracteres. """

    cad = input(print("Introduzca una cadena de texto:"))
    if (len(cad) > 5): return True
    else: return False
    print ("La cadena introducida es:", cad)      # Esta sentencia nunca se ejecutará

if (cadena_larga()): print("Supera los 5 caracteres")
else: print("No supera los 5 caracteres")
```

- Se introduce una cadena de texto por teclado que se almacena en la variable *cad.*
- Si la longitud del contenido de *cad* es de más de 5 caracteres se devuelve *True*. En caso contrario se devuelve *False.*
- El *print* de la función nunca llegará a ejecutarse ya que la función siempre se saldrá antes por alguno de los *return*.

- Devolución de valores (iii)

Se pueden devolver varios valores separándolos por comas.

```
# Ejemplo 10

def desglose(total):
    """    Desglosa un importe total en base imponible e IVA (21%).    """
    base = total / 1.21
    iva = total - base
    return base, iva

importe, impuestos = desglose(100)
print("El importe es:", importe, "y el IVA:", impuestos)
```

- La función *desglose()* devuelve dos valores en dos variables a través de la instrucción *return base,iva*
- Las variables importe e impuestos almacenarán los valores devueltos respectivamente.

- Devolución de valores (iv)

Se puede devolver más de un valor a través de una lista de valores.

```
# Ejemplo 11

def desglose(total):
    """    Desglosa un importe total en base imponible e IVA (21%).    """
    res = []
    res.append(total / 1.21)
    res.append(total - res[0])
    return res

resultado = desglose(100)
print("El importe es:", resultado[0], "y el IVA:", resultado[1])
```

- La función *desglose()* devuelve una lista de dos valores a través de la instrucción *return res.*
- La variable resultado recibe los valores devueltos por la función.
- La posición 0 de resultado contiene el importe y la posición 1 el IVA.

- Mutabilidad e inmutabilidad (i)

En Python todos los tipos de datos son objetos, pero no todos se comportan igual.

Objetos Inmutables

- Permite asignar nuevos valores.
- No permite modificar su contenido.
- enteros, flotantes, cadenas, tuplas.

Objetos Mutables

- Permite asignar nuevos valores.
- Permite modificar su contenido.
- listas, diccionarios, conjuntos.

- Mutabilidad e inmutabilidad (ii)

Si se intenta modificar el contenido de la zona donde apunta un objeto inmutable se producirá una excepción.

```
# Ejemplo 12

cadena = "hola"
cadena = cadena + " mundo"
print(cadena)
cadena[3] = "o"
print(cadena)
```

- El primer *print* imprimirá en pantalla la cadena de texto *"Hola mundo"*
- El segundo *print* no llegará a ejecutarse debido a que se producirá la excepción

  "TypeError: 'str' object does not support item assignment"

  al intentar ejecutar la instrucción *cadena[3]="o"* ya que un string es inmutable.

Esto no sucede en el caso de objetos mutables.

```
# Ejemplo 13

lista = [1, 2, 3]
print(lista)
lista.append(4)
lista[1] = 7
print(lista)
```

- El primer *print* imprimirá en pantalla *"[1, 2, 3]"*
- La instrucción *lista.append(4)* añadirá el número 4 al final de *lista.* Esta operación es posible debido a que una lista es un objeto mutable.
- La instrucción *lista[1] = 7* cambia el valor del segundo elemento de *lista*, es decir, cambia el *2* por un *7.* Esta operación es posible debido a que una lista es un objeto mutable.
- El segundo *print* imprime en pantalla *"[1, 7, 3, 4]"*

- Mutabilidad e inmutabilidad (iii)

Si varias variables apuntan a un mismo dato mutable y se produce un cambio sobre dicho dato, todas las variables verán reflejado ese cambio.

```
# Ejemplo 14

lista1 = [1, 2, 3]
lista2 = lista1
print(lista1)
print(lista2)
lista2.append(4)
print(lista1)
print(lista2)
```

- Inicialmente las variables lista1 y lista2 apuntan a los mismos datos y los dos primeros *print* imprimirán *[1, 2, 3]*
- La instrucción *lista2.append(4)* añadirá el número 4 al final de *lista2.* Esta operación es posible debido a que una lista es un objeto mutable.
- Los dos últimos *print* imprimen en pantalla *"[1, 2, 3, 4]".* Esto es debido a que las dos variables *lista1* y *lista2* apuntan a los mismos datos y, en consecuencia, cualquier operación sobre dichos datos afecta a ambas variables.

- Mutabilidad e inmutabilidad en los parámetros de una función (i)

También hay que tener en cuenta la mutabilidad o inmutabilidad de los argumentos en la invocación a una función.

En Python cuando se envía una variable como argumento en una llamada a una función lo que se envía es la referencia al objeto al que hace referencia la variable. Dependiendo de si el objeto es mutable o inmutable, la función podrá modificar o no el objeto.

```
# Ejemplo 15

def duplicar_valor(numero):
    """ Duplica el valor recibido. """
    numero = numero * 2
    print("Duplicado:", numero)

numero = 5
duplicar_valor(numero)
print("Despues de duplicar:", numero)
```

- La llamada a la función *duplicar_valor* se hace con el argumento de tipo entero 5.
- Al recibir la función una variable de tipo entero, lo que hace es que crea una nueva variable local *numero* apuntando al mismo sitio.
- Ya en el interior de la función se hace que la variable local *numero* apunte al valor entero 10 a través de la instrucción de asignación *numero = numero * 2* y luego se imprimirá el mensaje *"Duplicado: 10"* en pantalla.
- El *print* del cuerpo principal imprimirá el mensaje *"Despues de duplicar: 5"*. Esto se debe a que no se modificó el valor de la variable *numero* del cuerpo principal ya que es un objeto inmutable.

- Mutabilidad e inmutabilidad en los parámetros de una función (ii)

```
# Ejemplo 16

def duplicar_valor(numeros):
    """ Duplica el valor de los numeros recibidos. """
    for i in range(len(numeros)):
        numeros[i] = numeros[i] * 2

numeros = [5, 3]
print("Antes de duplicar:", numeros)
duplicar_valor(numeros)
print("Despues de duplicar:", numeros)
```

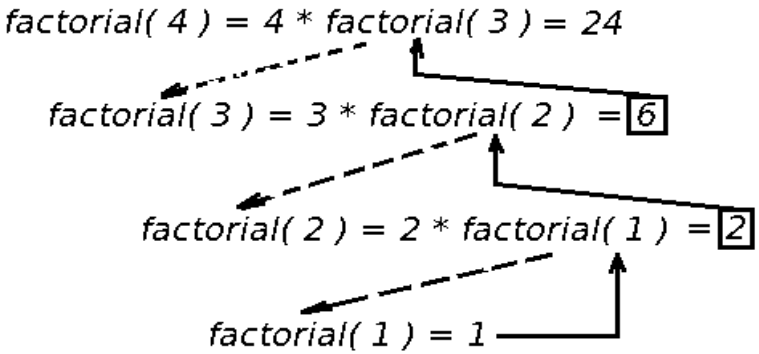
- La llamada a la función *duplicar_valor* se hace con el argumento de tipo lista *numeros*.
- Al recibir la función una variable de tipo lista, lo que hace es que crea una nueva variable local *numeros* apuntando al mismo sitio.
- En el interior de la función se duplica cada valor de la lista.
- El primer *print* del cuerpo principal imprimirá el mensaje *"Antes de duplicar: [5, 3]"*.
- El segundo *print* del cuerpo principal imprimirá el mensaje *"Despues de duplicar: [10, 6]*. Esto es debido a que una lista es un objeto mutable y, por tanto, lo que la función modificó fueron los datos que estaban apuntados por ambas variables (la local y la del cuerpo principal).

- Recursividad (i)

Se dice que una función es recursiva cuando se llama a sí misma durante su propia ejecución hasta llegar a uno de los *casos base (caso con solución directa)*.

```
# Ejemplo 17

def factorial(n):
 """ Calcula el factorial recursivo de n. """
   if (n < 0): return -1
   elif (n == 0 or n == 1): return 1          # casos base
   else: return (n * factorial(n-1))          # llamada recursiva
```



Todas los algoritmos recursivos tienen su equivalente iterativo, que es más eficiente.

```
# Ejemplo 18

def factorial(n):
   """ Calcula el factorial iterativo de n.   """
   if (n < 0): return -1
   resultado = 1
   for i in range(1,n+1):
      resultado *= i
   return resultado
```

- Ejercicios.

1. Define la función *area_circulo(r)* que devuelva el área de un círculo a partir de su rádio. (El área de un círculo se puede calcular a través de la fórmula: $a = \pi r^2$
2. Define la función *far2cen(f)* que devuelva el equivalente en grados Centígrados de un número *f* de grados Farenheit. (c = (f – 32) * 5/9)
3. Define la función *mayor_edad(edad)* que compruebe la mayoría de edad de una persona.
4. Defina la función *nif(dni)* que devuelva la letra del NIF correspondiente a un determinado DNI. Para ello se dividirá el dni por 23 y el resto de dicha división se hará corresponder con la siguiente tabla:

RESTO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LETRA	T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

5. Define la función *mi_división(dividendo, divisor)* que devuelva el cociente y el resto de la división entera dividendo/divisor.
6. Define la función *minmax(lista_valores)* que devuelva el valor mínimo y máximo de una lista de valores numéricos.
7. Define la función *ceros(lista_valores, umbral)*, de tal forma que ponga a cero todos los valores de la lista de valores menores que el umbral.
8. Define la función *leg(lista_valores, umbral)* que devuelva los elementos menores que *umbral*, los elementos iguales que *umbral* y los elementos mayores que *umbral*.
9. Define la función *par_impar(lista_valores)* que devuelva los e*lementos_pares* y los *elementos_impares* por separado.
10. Escribir la función recursiva *fib(n)* para calcular el valor del n-esimo término de la sucesión de Fibonacci (0,1,1,2,3,5, …). Donde: fib(0) = 0; fib(1) = 1; fib(n) = fib(n-1) + fib(n-2) para n > 1.

FUNCIONES

1. Define la función *area_circulo(r)* que devuelva el área de un círculo a partir de su rádio. (El área de un círculo se puede calcular a través de la fórmula: $a = \pi r^2$)

```
def area_circulo(r):
    """
    Parametros: r = radio
    Devuelve el area de un circulo a partir de su radio: a = pi * r * r
    Devuelve -1 si r es negativo
    """
    pi = 3.1416
    if(r < 0): return -1
    return ( pi * r * r)
```

2. Define la función *far2cen(f)* que devuelva el equivalente en grados Centígrados de un número *f* de grados Farenheit. (c = (f – 32) * 5/9)

```
def far2cen(f):
    """
    Devuelve el equivalente en grados Centígrados de un número f de grados Farenheit.
    c = (f – 32) * 5/9
    """
    return (f -32) * 5/9
```

3. Define la función *mayor_edad(edad)* que compruebe la mayoría de edad de una persona.

```
def mayor_edad(edad):
    """
    Comprueba la mayoria de edad de una persona.
    Devuelve True si la edad introducida >= 18
    Devuelve False si la edad introducida < 18.
    """
    return (edad >= 18)
```

4. Defina una función que devuelva la letra del NIF correspondiente a un determinado DNI. Para ello se dividirá el dni por 23 y el resto de dicha división se hará corresponder con la siguiente tabla:

RESTO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LETRA	T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

```
def nif(dni):
    """
    Devuelve la letra correspondiente a un dni.
    """
    letras = 'TRWAGMYFPDXBNJZSQVHLCKE'
    return letras[dni % 23]
```

5. Define la función *mi_división(dividendo, divisor)* que devuelva el cociente y el resto de la división entera dividendo/divisor.

```
def mi_division(dividendo, divisor):
    """
    Parametros: dos números flotantes: dividendo y divisor
    Realiza la division entera y devuelve una lista con cociente y resto
    Devuelve una lista con dos elementos: división entera y resto
    """
    return [dividendo // divisor, dividendo % divisor]
```

6. Define la función *minmax(lista_valores)* que devuelva el valor mínimo y máximo de una lista de valores numéricos.

```
def minmax(valores):
    """
    Recibe como argumento una lista de valores de tipo numerico
    Devuelve el menor y el mayor de los elementos de la lista de valores
    """
    return(min(valores), max(valores))
```

7. Define la función *ceros(lista_valores, umbral)*, de tal forma que ponga a cero todos los valores de la lista menores que el *umbral* (por defecto *umbral = 0.5)*.

```
def ceros(lista, umbral = 0.5):
    """    Parametros: lista valores, umbral
    Pone a cero todos los valores de la lista menores que un umbral. Por defecto el umbral es 0.5   """
    for i in range(len(lista)):
        if (lista[i] < umbral): lista[i] = 0
    return lista
```

8. Define la función *divide_lista(lista_valores, umbral)* que devuelva los elementos menores que *umbral*, los elementos iguales que *umbral* y los elementos mayores que *umbral*.

```
def divide_lista(lista, umbral):
    """
    Parametros: una lista de valores y un umbral (0.5 por defecto)
    Devuelve una lista segun este formato: [menores_que_umbral, iguales_que_umbral, mayores_que_umbral]
    """
    menores, iguales, mayores = [], [], []
    for i in range(len(lista)):
        if (lista[i] < umbral):    menores.append(lista[i])
        elif (lista[i] == umbral):
            iguales.append(lista[i])
        else:
            mayores.append(lista[i])
    res=menores, iguales, mayores
    return res
```

```
def divide_lista(lista, umbral):
    """
    Parametros: una lista de valores y un umbral (0.5 por defecto)
    Devuelve una lista segun este formato: [menores_que_umbral, iguales_que_umbral, mayores_que_umbral]
    """
    menores = [x for x in lista if x < umbral]
    iguales = [x for x in lista if x == umbral]
    mayores = [x for x in lista if x > umbral]
    res = menores,iguales,mayores
    return res
```

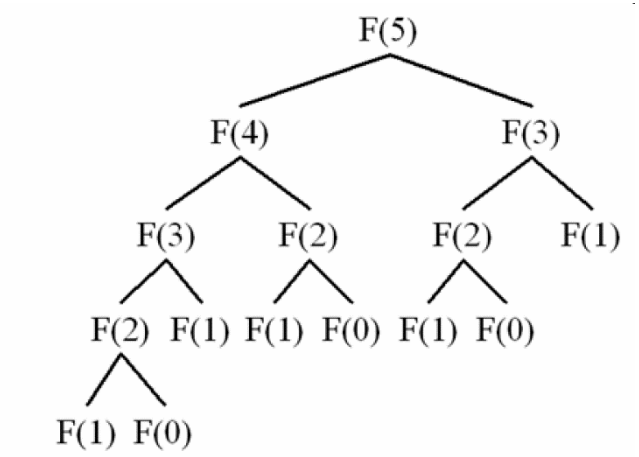
9. Define la función *par_impar(lista_valores)* que devuelva los e*lementos pares* y los *elementos impares* por separado.

```
def par_impar(lista):
    """
    Calcula los elementos pares e impares de una lista de valores
    Recibe: lista valores numericos
    Devuelve una lista con dos posiciones: pares e impares.
    """
    pares, impares = [], []
    for i in range(len(lista)):
        if (lista[i] % 2 == 0):
            pares.append(lista[i])
        else:
            impares.append(lista[i])
    res = pares,impares
    return res
```

```
def par_impar(lista):
    """
    Calcula los elementos pares e impares de una lista de valores
    Recibe: lista valores numericos
    Devuelve una lista con dos posiciones: pares e impares.
    """
    pares = [x for x in lista if (x % 2 == 0)]
    impares = [x for x in lista if (x % 2 != 0)]
    res = pares,impares
    return res
```

10. Escribir una función recursiva para calcular el valor del n-esimo término de la sucesión de Fibonacci (0,1,1,2,3,5, …).

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$




```
# Ejercicio 10


def fib_rec(n):
    """
    Calculo del valor del n-esimo termino de la sucesión de Fibonacci de forma recursiva
    f(0) = 0; f(1) = 1; f(n) = f(n-1) + f(n-2)
    """
    if n == 0: return 0         # caso base
    elif n == 1: return 1       # caso base
    else: return fib_rec(n-1) + fib_rec(n-2)    # caso recursivo
```