

# Resumen BÁSICO de FUNCIONES en Python

Javier Fernández Panadero

Noviembre 2022



Si os ha gustado y queréis contribuir a su difusión y otros contenidos, se agradece un ko-fi



# ÍNDICE

ÍNDICE	1
¿Qué es una función?	2
¿Para qué sirve?	2
Estructura	2
Ejemplo 1 (Sin parámetros, Sin retorno)	2
Ejemplo 2 (Con un parámetro, Sin retorno)	3
Ejemplo 3 (Con DOS parámetros, Sin retorno)	3
¿Quién programa las funciones?	4
Posibles problemas	4
Soluciones	4
Ejemplo 4 (Con DOS parámetros, Con retorno)	5
Ejemplo 5 (Devolviendo una expresión)	5
Ejemplo 6 (Varios return en una misma función)	6
Ejemplo 7 (Variables LOCALES y GLOBALES)	7
Ejemplo 8 (Número de parámetros indefinido —ordenados)	7
Ejemplo 9 (Parámetros con valores por defecto)	8
Ejemplo 10 (Parámetros desordenados)	8
Ejemplo 11 (Devuelve dos valores para guardar en dos variables)	9
Mejorar la ayuda (Tipos de entrada y salida)	9
Ejemplo 12 (Número de parámetros indefinido —desordenados)	10
Args y Kwargs, resumen	10
Las funciones de Python	11
Funciones de “otros”. Importación de módulos	12
Ejemplo 13 (Importación de un módulo y uso de función)	13
Ejemplo 14 (Importar sólo un elemento de un módulo)	14
En los módulos hay más “cosas”.	15
Hasta aquí...	15

## ¿Qué es una función?

Unas cuantas instrucciones que agrupamos para que se ejecuten a la vez cuando son “invocadas”.

## ¿Para qué sirve?

Para simplificar los programas

- No se repite código (se escribe una vez y se llama las que se quiera)
- El programa queda en “partes” más fáciles de escribir, leer, mantener o corregir

## Estructura

```
def nombre_funcion(parametros):  
    """Docstring, explicación de qué hace"""  
    instrucciones  
    return valores_que_devuelve
```

## Ejemplo 1 (Sin parámetros, Sin retorno)



```
main.py x +  
1 def dibuja_rectangulo1():  
2     """Dibuja un rectángulo con arrobas"""  
3     print('@@@@@@')  
4     print('@@@@@@')  
5     print('@@@@@@')  
6  
7     dibuja_rectangulo1()  
8     print()  
9     dibuja_rectangulo1()  
10    print()  
11    dibuja_rectangulo1()  
12  
13  
Console x  
@@@@@@  
@@@@@@  
@@@@@@  
  
@@@@@@  
@@@@@@  
@@@@@@  
  
@@@@@@  
@@@@@@  
@@@@@@
```

Fíjate:

- Empezamos poniendo `def` y terminamos con dos puntos
- Hay que poner los paréntesis aunque no haya parámetros
- Las instrucciones van indentadas
- Hemos dibujado tres rectángulos llamando a la función, pero solo ha habido que definirla una vez. Por eso ahorramos código.

## Ejemplo 2 (Con un parámetro, Sin retorno)



```
1 def dibuja_rectangulo3(largo):
2     """Dibuja un rectángulo con arrobas de un
   determinado largo"""
3     print('@' * largo)
4     print('@' * largo)
5     print('@' * largo)
6
7     dibuja_rectangulo3(3)
8     print()
9     dibuja_rectangulo3(6)
10    print()
11    dibuja_rectangulo3(9)
12
```

Console output:

```
@@@
@@@
@@@

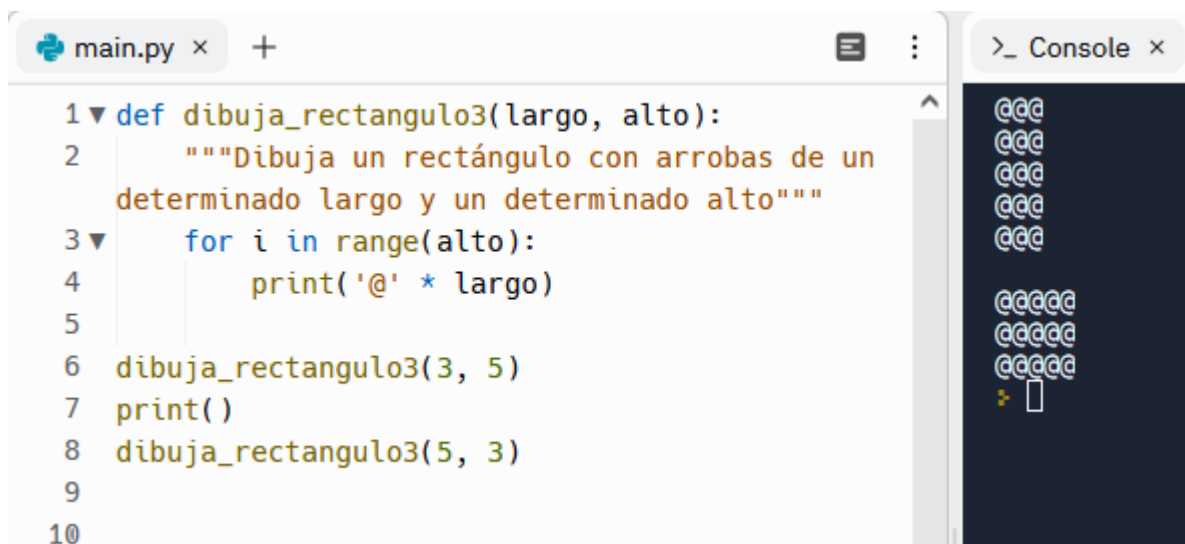
@@@@@@
@@@@@@
@@@@@@

@@@@@@@@@@
@@@@@@@@@@
@@@@@@@@@@
```

Fíjate:

- El nombre del parámetro se usa en la definición para ver en qué lugares de la función va a aparecer después. En nuestro caso es el número de veces que repetimos el carácter arroba.
- Cuando se llama a la función hay que darle el valor del parámetro, si no dará error.

## Ejemplo 3 (Con DOS parámetros, Sin retorno)



```
1 def dibuja_rectangulo3(largo, alto):
2     """Dibuja un rectángulo con arrobas de un
   determinado largo y un determinado alto"""
3     for i in range(alto):
4         print('@' * largo)
5
6     dibuja_rectangulo3(3, 5)
7     print()
8     dibuja_rectangulo3(5, 3)
9
10
```

Console output:

```
@@@
@@@
@@@
@@@
@@@

@@@@@
@@@@@
@@@@@
@@@@@
@@@@@

@@@
@@@
@@@
```

Fíjate:

- Cada parámetro se usa para una cosa
- Los parámetros están ordenados: el primer número que demos será el largo. El segundo, el alto.

## ¿Quién programa las funciones?

- Definidas por nosotros: como los ejemplos anteriores)
- Las propias de Python: como `print()`, `input()`, `len()`, etc.
- Definidas por otros. Tenemos que “importarlas” y después podemos usarlas

## Posibles problemas

- Las funciones que no definimos nosotros, no sabemos lo que hacen por dentro.
- Es peligroso que impriman lo que quieran, que cambien valores de variables de otras partes del programa...

## Soluciones

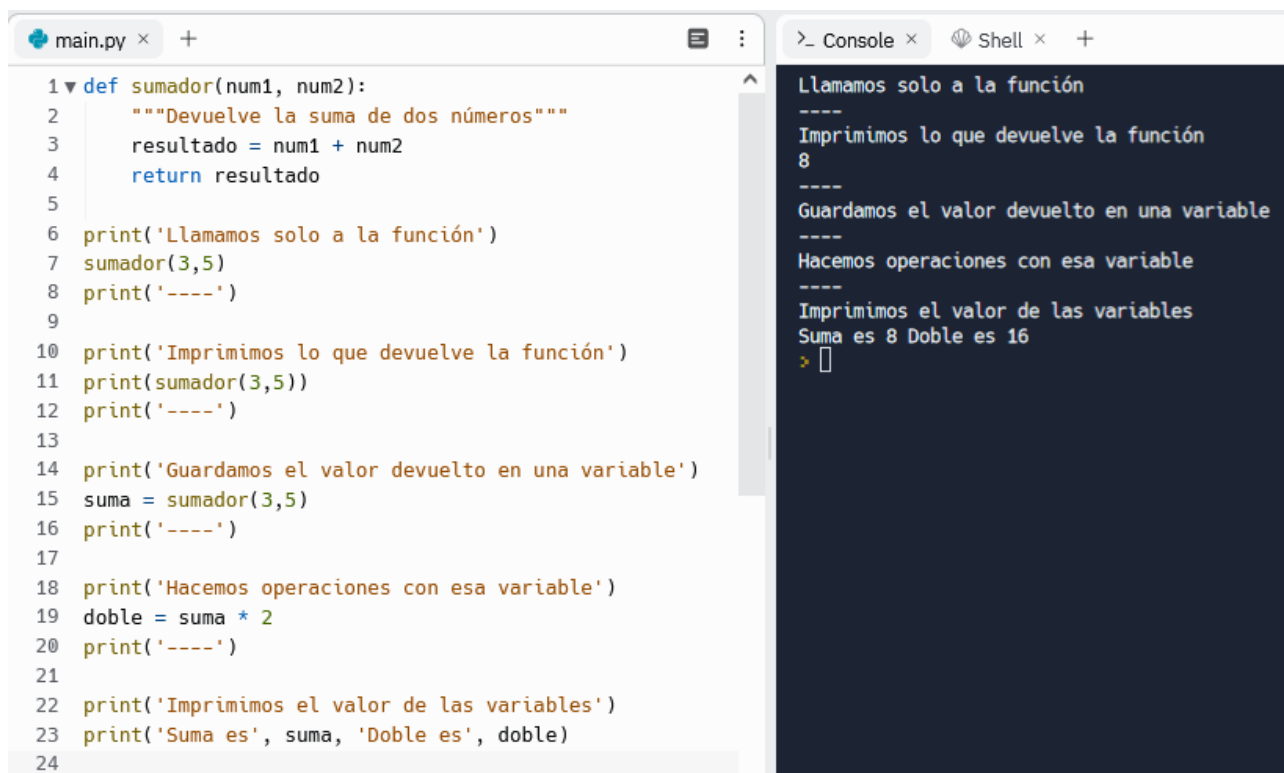
Lo ideal es programarlas de manera:

- Que tomen valores de entrada y DEVUELVAN valores de salida. Nada más.
- Que cada función haga solo una cosa
- Que dados los mismos parámetros, devuelva el mismo valor (esté como esté el resto del programa: otras variables, en qué momento de su ejecución, etc.)

Así que, a partir de aquí nos ceñiremos a esto lo más posible.

Cabe comentar que NO hay funciones sin retorno. Las que no tienen explícitamente un *return* devuelven `None`.

## Ejemplo 4 (Con DOS parámetros, Con retorno)



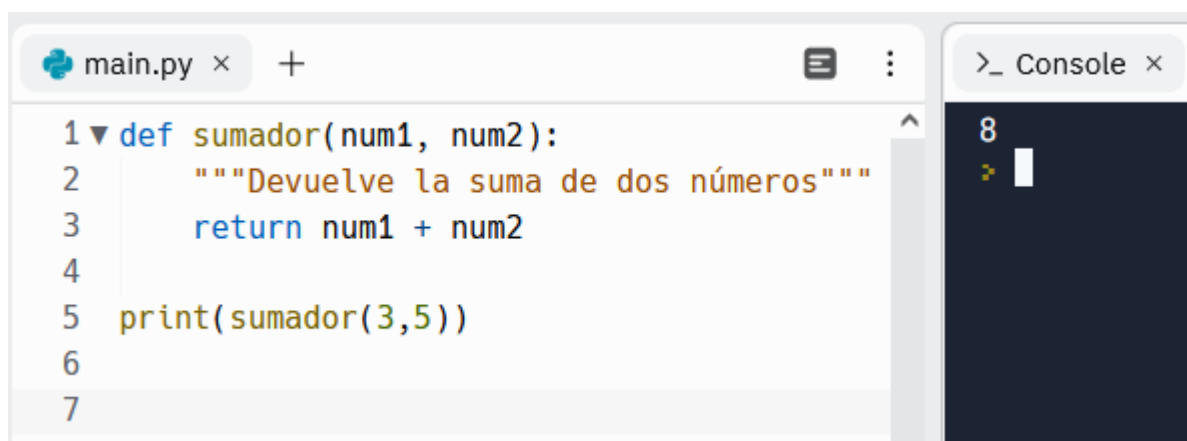
```
main.py x +
1 def sumador(num1, num2):
2     """Devuelve la suma de dos números"""
3     resultado = num1 + num2
4     return resultado
5
6 print('Llamamos solo a la función')
7 sumador(3,5)
8 print('----')
9
10 print('Imprimimos lo que devuelve la función')
11 print(sumador(3,5))
12 print('----')
13
14 print('Guardamos el valor devuelto en una variable')
15 suma = sumador(3,5)
16 print('----')
17
18 print('Hacemos operaciones con esa variable')
19 doble = suma * 2
20 print('----')
21
22 print('Imprimimos el valor de las variables')
23 print('Suma es', suma, 'Doble es', doble)
24
```

```
>_ Console x Shell x +
Llamamos solo a la función
----
Imprimimos lo que devuelve la función
8
----
Guardamos el valor devuelto en una variable
----
Hacemos operaciones con esa variable
----
Imprimimos el valor de las variables
Suma es 8 Doble es 16
> |
```

Fíjate:

- El valor que se devuelve va precedido de la palabra *return*
- Si sólo llamas a la función no ocurre nada, tienes que hacer una de estas cosas:
  - Imprimir el valor que devuelva
  - Guardar el valor devuelto en alguna variable
- Si imprimes, ya lo ves.
- Si guardas, puedes usar ese valor para hacer operaciones
  - Más tarde puedes imprimirlo si lo quieres ver

## Ejemplo 5 (Devolviendo una expresión)



```
main.py x +
1 def sumador(num1, num2):
2     """Devuelve la suma de dos números"""
3     return num1 + num2
4
5 print(sumador(3,5))
6
7
```

```
>_ Console x
8
> |
```

Fíjate:

- *return* puede devolver el resultado de una expresión, no solo valores de variables
- Cuando la expresión es sencilla podemos reducir código sin que pierda legibilidad

## Ejemplo 6 (Varios return en una misma función)



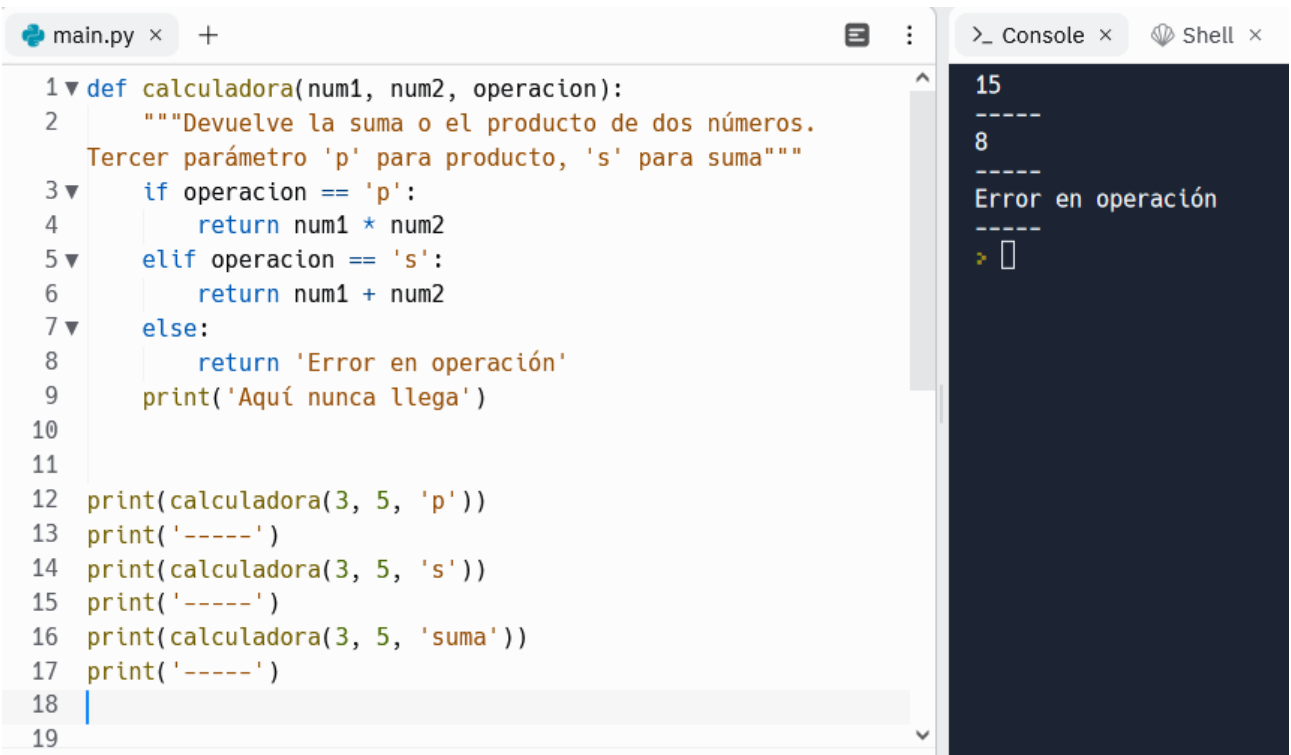
```
1 def calculadora(num1, num2, operacion):
2     """Devuelve la suma o el producto de dos números. Tercer parámetro 'p' para producto, 's' para suma"""
3     if operacion == 'p':
4         return num1 * num2
5     elif operacion == 's':
6         return num1 + num2
7     else:
8         return 'Error en operación'
9
10
11
12
13
14
15 print(calculadora())
16
17
```

Tooltip text:

```
calculadora(num1, num2, operacion)
Devuelve la suma o el producto de dos números.
Tercer parámetro 'p' para producto, 's' para suma
```

Fíjate:

- Una de las funciones del Docstring es servir de ayuda al escribir la función. Aquí me avisa de qué significan los valores y qué tengo que poner.



```
1 def calculadora(num1, num2, operacion):
2     """Devuelve la suma o el producto de dos números.
3     Tercer parámetro 'p' para producto, 's' para suma"""
4     if operacion == 'p':
5         return num1 * num2
6     elif operacion == 's':
7         return num1 + num2
8     else:
9         return 'Error en operación'
10     print('Aquí nunca llega')
11
12 print(calculadora(3, 5, 'p'))
13 print('-----')
14 print(calculadora(3, 5, 's'))
15 print('-----')
16 print(calculadora(3, 5, 'suma'))
17 print('-----')
18
19
```

Console output:

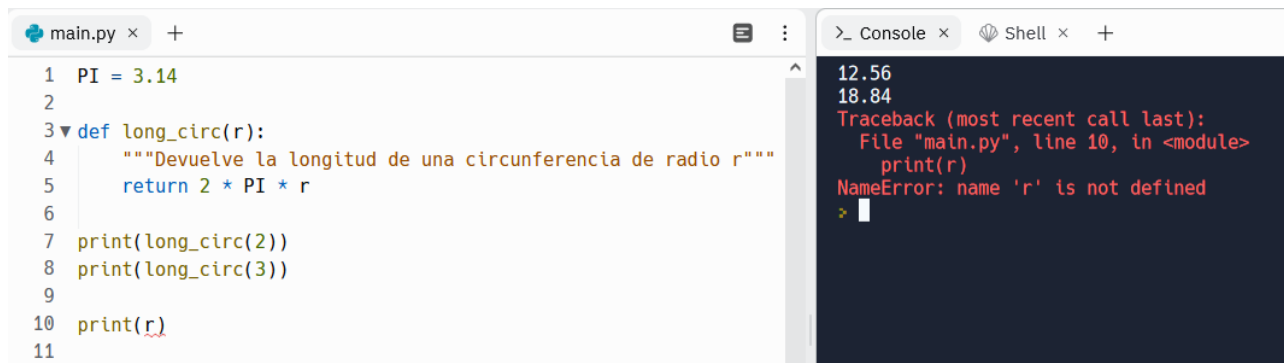
```
15
-----
8
-----
Error en operación
-----
>
```

Fíjate:

- Cuando se llega a algún *return* la función devuelve el valor y se termina. Ninguna de las ejecuciones de la función llega al *print* del final.

## Ejemplo 7 (Variables LOCALES y GLOBALES)

- Los parámetros y las variables que se definen dentro de las funciones sólo existen dentro de la función.
- Variables o constantes que existen fuera de la función SÍ PUEDEN usarse dentro de la función



The screenshot shows a Python IDE with a file named 'main.py' and a console window. The code in 'main.py' is as follows:

```
1 PI = 3.14
2
3 def long_circ(r):
4     """Devuelve la longitud de una circunferencia de radio r"""
5     return 2 * PI * r
6
7 print(long_circ(2))
8 print(long_circ(3))
9
10 print(r)
11
```

The console window shows the output of the first two print statements: 12.56 and 18.84. It then shows a traceback error:

```
Traceback (most recent call last):
  File "main.py", line 10, in <module>
    print(r)
NameError: name 'r' is not defined
>
```

Fíjate:

- Las constantes se escriben al principio del programa (después de los import)
- Las constantes se escriben con mayúsculas y guiones bajos
- El radio r deja de existir fuera de la función, es una variable LOCAL

## Ejemplo 8 (Número de parámetros indefinido —ordenados)



The screenshot shows a Python IDE with a file named 'main.py' and a console window. The code in 'main.py' is as follows:

```
1 def junta_letras(*letras):
2     """Devuelve las letras concatenadas"""
3     salida = ''
4     for letra in letras:
5         salida += letra
6     return salida
7
8 print(junta_letras('a', 'c'))
9 print(junta_letras('a', 'c', 'j'))
10 print(junta_letras('a', 'c', 'u', 't'))
11
12
```

The console window shows the output of the three print statements: ac, acj, and acut.

Fíjate:

- El asterisco nos dice que el número de parámetros puede ser cualquiera
- Con frecuencia se pone \*args (de arguments) pero lo importante es el asterisco, puedes llamar a ese “grupo” de parámetros con cualquier nombre
- Con un bucle o cualquier otra manera de recorrer letras puedo ir usando los parámetros que se hayan introducido. Esto es porque se han guardado en una tupla, que es un iterable. (una tupla es como una lista, pero inmutable).
- El orden de los parámetros sigue siendo importante.



## Ejemplo 9 (Parámetros con valores por defecto)



```
1 def apuntar(nombre, pago = 0):
2     """Devuelve un diccionario con el nombre y el saldo"""
3     return {'nombre': nombre, 'saldo': pago}
4
5 print(apuntar('Javi'))
6 print(apuntar('Javi', 52))
7
8
```

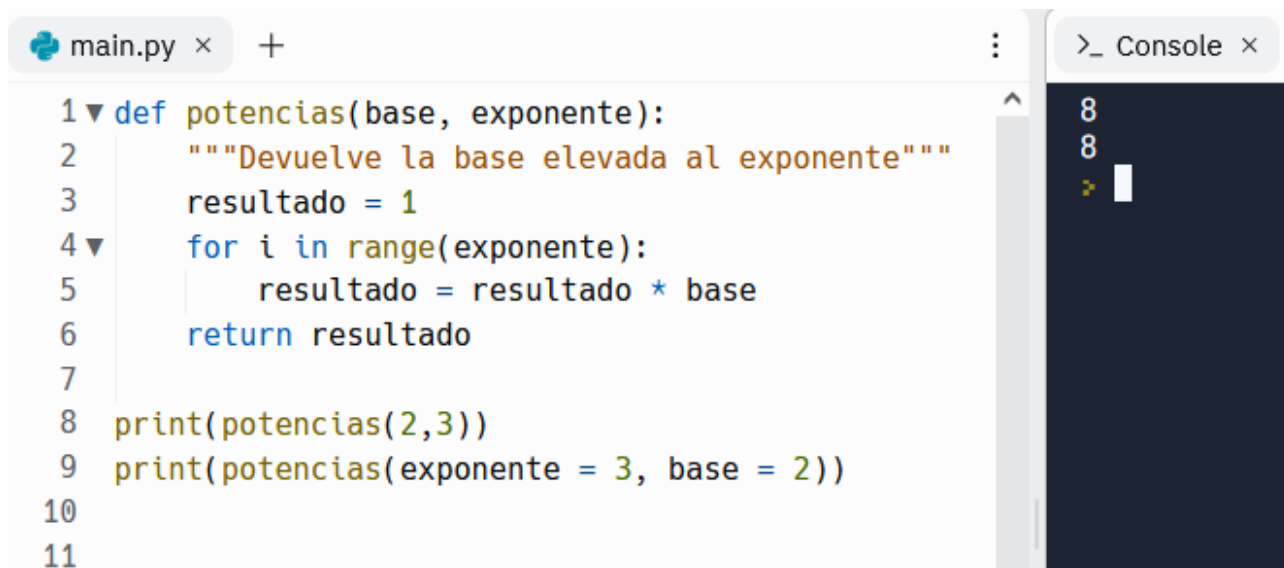
```
{'nombre': 'Javi', 'saldo': 0}
{'nombre': 'Javi', 'saldo': 52}
```

Fíjate:

- Cuando un parámetro tiene un valor por defecto
  - Puedo no incluirlo en la llamada y tendrá ese valor
  - Puedo incluirlo y ponerle el valor que quiera
- Una función puede devolver cualquier tipo de datos. En este caso, un diccionario.
- Recuerda: Una función también admite cualquier tipo de datos como parámetros

## Ejemplo 10 (Parámetros desordenados)

- En los ejemplos anteriores sabíamos qué valor era para qué parámetro por el orden en que se escribían.
- Podemos usar la sintaxis “key-value” (clave-valor) y así NO importa el orden en el que se pongan



```
1 def potencias(base, exponente):
2     """Devuelve la base elevada al exponente"""
3     resultado = 1
4     for i in range(exponente):
5         resultado = resultado * base
6     return resultado
7
8 print(potencias(2,3))
9 print(potencias(exponente = 3, base = 2))
10
11
```

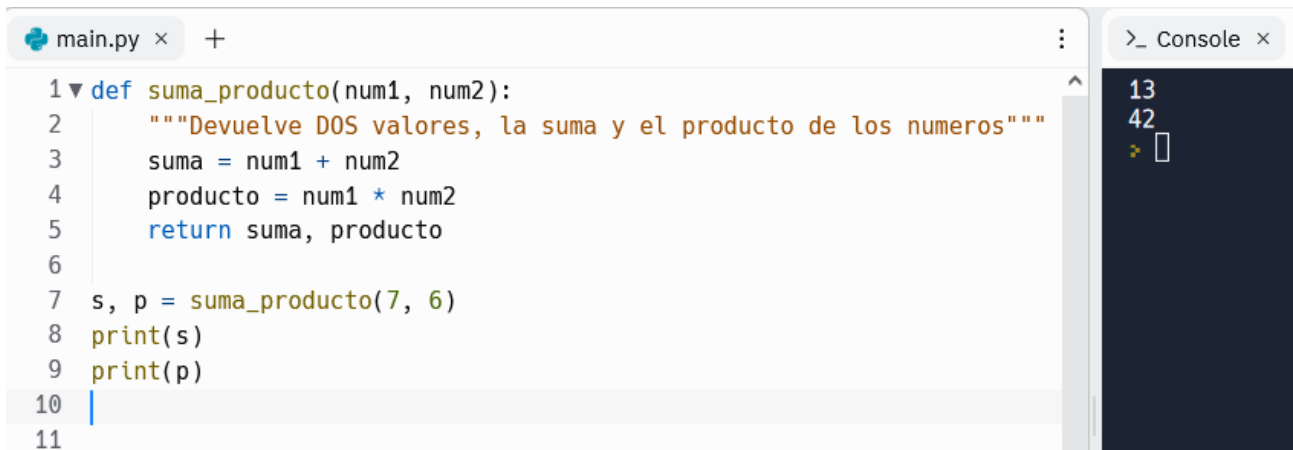
```
8
8
```

Fíjate:

- En el segundo caso se han pasado los parámetros en orden diferente a como están en la definición, pero como se decía qué valor era para cada parámetro, la función da el resultado adecuado.
- Esto es cómodo, no tengo que recordar el orden de los parámetros, sólo ir pasándolos en parejas *key = value* (nombre = 'Javi', ciudad = 'Pinto', job = 'profesor'...)

## Ejemplo 11 (Devuelve dos valores para guardar en dos variables)

- Para devolver “grupos” de valores con un *return* podríamos usar varias tácticas como, por ejemplo, devolver una lista con los valores.
- Es una práctica común hacerlo de la siguiente forma.



```
1 def suma_producto(num1, num2):
2     """Devuelve DOS valores, la suma y el producto de los numeros"""
3     suma = num1 + num2
4     producto = num1 * num2
5     return suma, producto
6
7 s, p = suma_producto(7, 6)
8 print(s)
9 print(p)
10
11
```

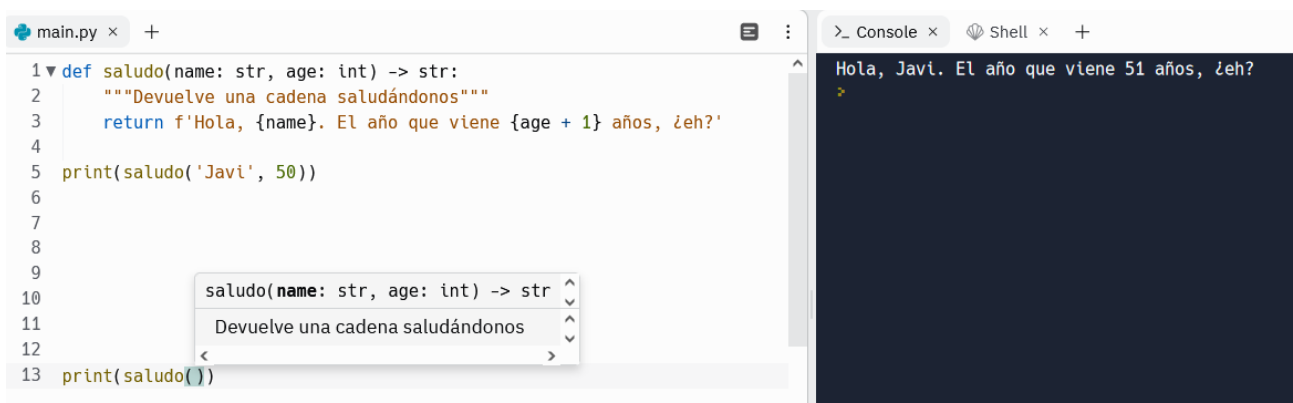
Console output:

```
13
42
```

Fíjate:

- Los valores devueltos están separados por comas
- En la asignación de lo que devuelve la función, también tengo que poner tantas variables como valores me devuelven (separados por comas)
- En realidad lo que se devuelve es UNA tupla, que se puede escribir (suma, producto) o suma, producto.
- Al proceso de asignar variables a cada elemento de una tupla (o una lista) se le llama *unpacking*

## Mejorar la ayuda (Tipos de entrada y salida)



```
1 def saludo(name: str, age: int) -> str:
2     """Devuelve una cadena saludándonos"""
3     return f'Hola, {name}. El año que viene {age + 1} años, ¿eh?'
4
5 print(saludo('Javi', 50))
6
7
8
9
10
11
12
13 print(saludo())
```

Tooltip:

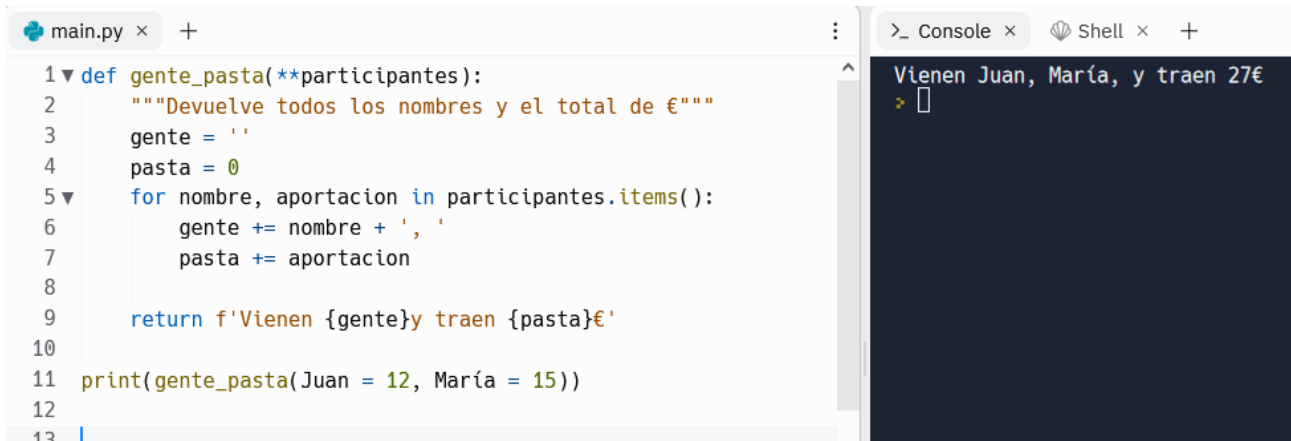
```
saludo(name: str, age: int) -> str
Devuelve una cadena saludándonos
```

Console output:

```
Hola, Javi. El año que viene 51 años, ¿eh?
```

- Así podemos enriquecer la ayuda de nuestra función, especificando tanto el tipo de cada variable de entrada como la indicación de que es una función con retorno y que tipo de datos va a devolver
- En nuestro caso:
  - (*name: str*) es que name es de tipo cadena de caracteres
  - (*age: int*) es que age es de tipo entero
  - (*-> str*) es que va a devolver algo, y que será una cadena de caracteres

## Ejemplo 12 (Número de parámetros indefinido —desordenados)



```
1 def gente_pasta(**participantes):
2     """Devuelve todos los nombres y el total de €"""
3     gente = ''
4     pasta = 0
5     for nombre, aportacion in participantes.items():
6         gente += nombre + ', '
7         pasta += aportacion
8
9     return f'Vienen {gente}y traen {pasta}€'
10
11 print(gente_pasta(Juan = 12, María = 15))
12
13
```

Vienen Juan, María, y traen 27€

### Fíjate

- Hay dos asteriscos para indicar que los parámetros se van a dar como key-value
- Puedes llamar al “grupo” de parámetros como quieras, se suele usar kwargs (keyword arguments)
- Al dar los argumentos así, se guardan como un diccionario (con args se guardaban como una tupla)
- Puedo recorrer el diccionario como quiera para ir usando los valores de los parámetros. En este caso, he querido usar el método `.items()` para sacar tanto la clave como el valor y por eso después del `for` hay dos variables, una para guardar la clave y otra para guardar el valor.
- Quizá te resulte curioso que el símbolo “+” vale tanto como operador de concatenación de *strings* y como operador de sumar enteros.
- En el `return` hemos usado la más reciente forma de dar resultados con mezcla strings literales y valores de variables. Es sólo poner `f` primero, todo entre comillas y los valores de variables entre llaves.

## Args y Kwargs, resumen

- En las funciones podemos tener parámetros fijos posicionales, parámetros de número indefinido ordenados (args) y parámetros por clave-valor (kwargs)
- Por ejemplo, en **`myfunction(variable1, variable2, *args, **kwargs)`**
- Al llamarla, los dos primeros valores irían a las variables uno y dos, los siguientes se irían guardando en una tupla y, los que se diesen por clave-valor, en un diccionario.
- Por ejemplo, si digo `myfunction(3, 7, 5, 2, 6, nombre = 'Juan', age = 34)` los parámetros se guardarían así:
  - `variable1 = 3`
  - `variable2 = 7`
  - `args = (5, 2, 6)`
  - `kwargs = {'nombre': 'Juan', 'age': 34}`

## Las funciones de Python

- Las funciones que incluye Python (built-in) podemos usarlas con toda libertad, como en realidad ya hacemos *print()*, *int()*, *input()*, *len()*, etc.
- Ahora podríamos fijarnos en características que no habíamos reparado antes, por ejemplo en la conocida función **print()**
- Si usamos la ayuda en la consola nos dice esto.

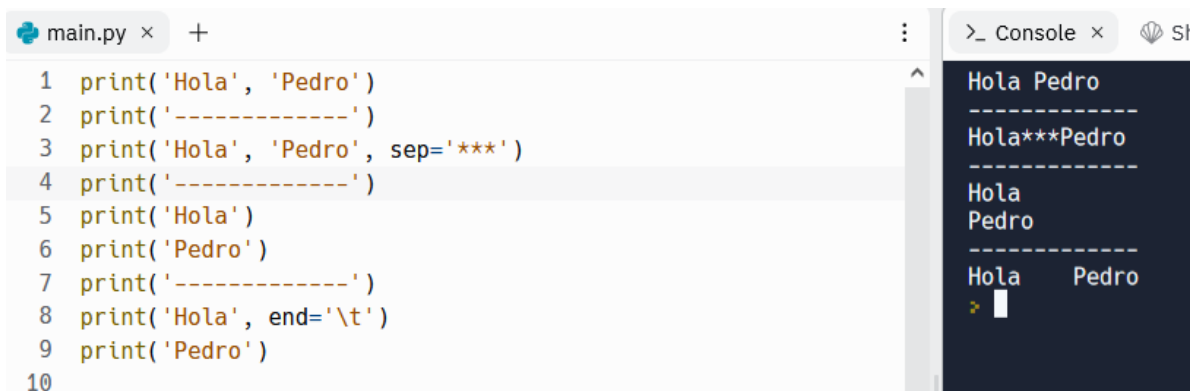
```
> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys
          s.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline
    flush: whether to forcibly flush the stream.
```

Fíjate:

- Puede imprimir tantos elementos como queramos (como con *\*args*)
- Hay varios argumentos de tipo keyword con valores por defecto, por ejemplo:
  - *sep*: separador entre cadenas impresas, por defecto un espacio.
  - *end*: qué se pone al final de cada impresión, por defecto un cambio de línea
- *\n* es cómo se simboliza el cambio de línea. Hay más códigos como este. Por ejemplo *\t* es una tabulación.



The screenshot shows a Python IDE with a file named `main.py`. The code in the editor is as follows:

```
1 print('Hola', 'Pedro')
2 print('-----')
3 print('Hola', 'Pedro', sep='***')
4 print('-----')
5 print('Hola')
6 print('Pedro')
7 print('-----')
8 print('Hola', end='\t')
9 print('Pedro')
10
```

The console on the right shows the output of the code:

```
Hola Pedro
-----
Hola***Pedro
-----
Hola
Pedro
-----
Hola    Pedro
>
```

Fíjate:

- Si usamos *print* sin especificar nada, separa *Hola* y *Pedro* por espacios, si le cambio el separador por una cadena de asteriscos, pues usa eso último.
- Si uso dos sentencias *print*, pone cada cosa en una línea, porque termina cada *print* con un cambio de línea.
- Si le especifico que termine con una tabulación, por ejemplo, pues ya ves que no hay cambio de línea aunque se usen dos sentencias *print*.

## Funciones de “otros”. Importación de módulos

- Podemos compartir las funciones que hagamos o bien usar las de otros.
- Como ya hemos dicho, “deben” ser *cajas negras* que toman parámetros y devuelven cosas sin interaccionar de otra forma o tener otros efectos.
- Así que podemos usar una función que nos pase de grados a radianes sin preocuparnos de nada, le pasamos los grados, recogemos los radianes que nos devuelva y ni siquiera necesitamos saber cómo está programada.
- Las funciones se “almacenan” en **MÓDULOS** y si nos hacemos con esos módulos, podremos usar sus funciones.
- Por ejemplo, **math** es un módulo lleno de funciones matemáticas, **random** contiene funciones relacionadas con “asuntos” aleatorios.
- Antes de poder usar un módulo hay que descargárselo de Internet, que te lo envíe un amigo, etc, ES UN FICHERO, por ejemplo: random.py
- Aunque Python ya tiene muchos módulos descargados, preparados para poder traerlos a tus programas y usarlos. Esa acción se llama **importar**.

### Importemos el módulo RANDOM



```
>_ Console x Shell x +
> import random
> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom',
 '_Sequence', '_Set', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__', 'accumulate', 'acos', '_bisect',
 '_ceil', '_cos', '_e', '_exp', '_inst', '_log', '_os', '_pi', '_random', '_repeat',
 '_sha512', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn', 'betavariate',
 'choice', 'choices', 'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate',
 'lognormvariate', 'normalvariate', 'paretovariate', 'randint', 'random', 'randrange',
 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
 'weibullvariate']
> help(random.randint)
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.

> 
```

Fíjate:

- import random nos “carga” el módulo random para que lo podamos usar
- dir(random) nos dice todas las funciones (y más) que hay en el módulo random
- help(random.randint) me da la ayuda sobre la función randint perteneciente al módulo random
- Podemos leer que nos devuelve un entero entre los valores que damos, inclusive.

## Ejemplo 13 (Importación de un módulo y uso de función)



The screenshot shows a code editor with a file named 'main.py'. The code defines a function 'dado6()' that uses 'random.randint(1,6)' to generate a random number between 1 and 6. A loop runs 10 times, calling 'dado6()' and printing the result. The console on the right shows the output: 1, 5, 5, 1, 2, 5, 3, 1, 6, 6.

```
1 import random
2 def dado6():
3     dado = random.randint(1,6)
4     return dado
5
6 for i in range(10):
7     print(dado6())
8
```

Console output: 1, 5, 5, 1, 2, 5, 3, 1, 6, 6

Fíjate:

- La importación de módulos es lo primero que se escribe en un programa.
- No podemos llamar directamente randint() porque se creería que es una función propia de Python y daría un error al no encontrarla.

A veces se usan **ALIAS** para los módulos porque tienen los nombres muy largos o porque se ha creado la costumbre.



The screenshot shows a code editor with a file named 'main.py'. The code imports 'random' as 'r' and defines a function 'dado6()' that uses 'r.randint(1,6)'. A loop runs 10 times, calling 'dado6()' and printing the result. The console on the right shows the output: 5, 4, 5, 5, 1, 4, 1, 4, 5, 5.

```
1 import random as r
2 def dado6():
3     dado = r.randint(1,6)
4     return dado
5
6 for i in range(10):
7     print(dado6())
8
```

Console output: 5, 4, 5, 5, 1, 4, 1, 4, 5, 5

Fíjate

- La línea de importación es diferente import random as r
- La llamada a la función es diferente r.randint()

Otros alias de módulos muy usados son:

import math as m

import numpy as np

import pandas as pd

## Ejemplo 14 (Importar sólo un elemento de un módulo)

Si sólo estás interesado en usar un elemento de un módulo no es necesario importar el módulo entero.

Aquí podemos ver el módulo `math` y la función raíz cuadrada

```
>_ Console x Shell x +
> import math
> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
> help(sqrt)
Help on built-in function sqrt in module math:

sqrt(x, /)
    Return the square root of x.
```

Importemos SOLO la función `sqrt()` y hagamos un programa

```
main.py x +
1 from math import sqrt
2 print(sqrt(16))
3 print(cos(3.14))

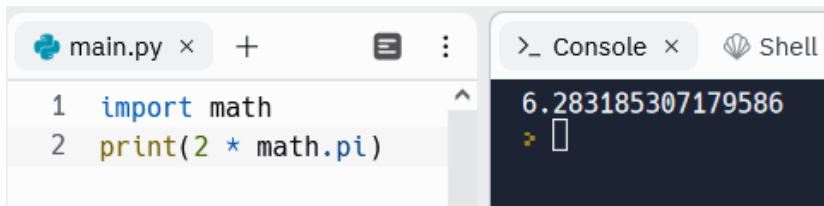
>_ Console x Shell x +
4.0
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print(cos(3.14))
NameError: name 'cos' is not defined
```

Fíjate

- Al importarla así NO hay que decir de qué módulo viene, se puede usar directamente solo con su nombre.
  - Ventajas: Comodidad
  - Desventaja: ¡¡Puedes haber “borrado” otra función de Python o que hubieras hecho tú que se llamara igual!!
  - Esto se puede hacer importando todas las funciones de un módulo. Está muy desaconsejado
- He intentado usar la función coseno (que puedes ver, dos imágenes más arriba, que también está en el módulo `math`), pero me ha dado error porque NO he importado todo el módulo `math`, sino sólo `sqrt()`.

En los módulos hay más “cosas”.

Por ejemplo, constantes. En math tienes, entre otras, los números:  $\pi$ , e.



The screenshot shows a code editor with a file named 'main.py' and a terminal window. The code in 'main.py' is:

```
1 import math
2 print(2 * math.pi)
```

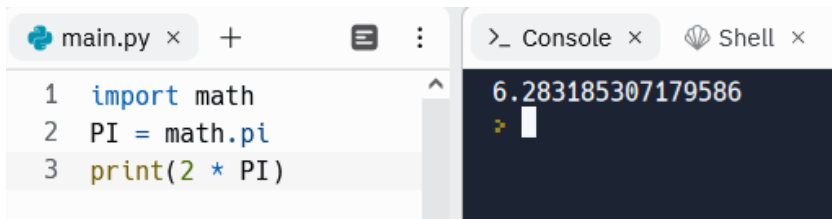
The terminal window shows the output of the script:

```
6.283185307179586
```

Fíjate que no van con paréntesis, porque no son funciones, son constantes.

También podríamos hacer esto, que quizá sea más cómodo.

(Recuerda, las constantes en mayúsculas)



The screenshot shows a code editor with a file named 'main.py' and a terminal window. The code in 'main.py' is:

```
1 import math
2 PI = math.pi
3 print(2 * PI)
```

The terminal window shows the output of the script:

```
6.283185307179586
```

Hasta aquí...

Este es un resumen de las características básicas de las funciones, hay mucho más, claro.

- Si te parece de utilidad se agradece la difusión de este trabajo, así como el apoyo [ko-fi](#) a quien pueda.
- Si ves alguna errata, incorrección o posible mejora, se agradece que nos lo digas.

Juntos somos más.