Funciones

Algoritmos y Programación Básica (CC2005) - 2026

Funciones

Semestre 02, 2026

El problema del código repetido

A medida que los programas crecen, aparecen bloques de código que se repiten.

Copiar y pegar el mismo código en varios lugares genera problemas.

Ejemplo

Un programa calcula el área de un rectángulo en tres lugares distintos.

```
# Primer lugar
area1 = base1 * altura1
print("Área:", area1)

# Segundo lugar
area2 = base2 * altura2
print("Área:", area2)

# Tercer lugar
area3 = base3 * altura3
print("Área:", area3)
```

El problema

- Si la fórmula cambia, hay que cambiarla en tres lugares.
- Si hay un error, hay que corregirlo en tres lugares.
- El programa se vuelve difícil de leer y mantener.

La solución

Escribir el código una sola vez y reutilizarlo cuantas veces sea necesario.

Eso es exactamente lo que hacen las funciones.

¿Qué es una función?

Una función es un bloque de código con nombre que realiza una tarea específica.

Se define una vez y se puede usar múltiples veces.

Funciones que ya se conocen

```
print("Hola")
input("Escribe algo: ")
int("42")
float("3.14")
```

Todas estas son funciones de Python.

Se usan escribiendo su nombre seguido de paréntesis.

Funciones propias

Además de las funciones integradas, se pueden crear funciones propias.

El programador decide qué hace la función, cómo se llama y qué recibe.

Definir una función

Para crear una función se usa la palabra clave `def`.

Sintaxis

```
def nombre_funcion():
    # bloque de código
```

- `def` indica que se está definiendo una función.
- `nombre_funcion` es el nombre que se le da a la función.
- Los dos puntos `:` abren el bloque.
- El bloque debe estar indentado.

Ejemplo

```
def saludar():
    print("¡Hola, bienvenido!")
```

Esto define la función. El código adentro no se ejecuta todavía.

Llamar a una función

Definir una función no la ejecuta.

Para ejecutarla hay que llamarla usando su nombre seguido de paréntesis.

Ejemplo

```
def saludar():
    print("¡Hola, bienvenido!")

saludar()
```

Resultado:

```
¡Hola, bienvenido!
```

Llamar múltiples veces

```
saludar()
saludar()
saludar()
```

Resultado:

```
¡Hola, bienvenido!
¡Hola, bienvenido!
¡Hola, bienvenido!
```

El mismo código se ejecutó tres veces sin repetirlo.

Parámetros

Un parámetro es una variable que la función recibe cuando se llama.

Permite que la función trabaje con datos distintos cada vez.

Sintaxis

```
def nombre_funcion(parametro):
    # usar parametro aquí
```

Ejemplo

```
def saludar(nombre):
    print("¡Hola,", nombre + "!")
```

`nombre` es el parámetro. Actúa como una variable dentro de la función.

Llamando con un argumento

```
saludar("María")
saludar("Carlos")
saludar("Ana")
```

Resultado:

```
¡Hola, María!
¡Hola, Carlos!
¡Hola, Ana!
```

- Al llamar `saludar("María")`, el parámetro `nombre` toma el valor `"María"`.
- Cada llamada pasa un argumento distinto.

Parámetro vs argumento

| Término | Descripción | Ejemplo | | --- | --- | --- | | Parámetro | Variable en la definición | `def saludar(nombre):` | | Argumento | Valor al llamar la función | `saludar("María")` |

Múltiples parámetros

Una función puede recibir más de un parámetro.

Se separan con comas.

Ejemplo

```
def sumar(a, b):
    resultado = a + b
    print("La suma es:", resultado)

sumar(3, 5)
sumar(10, 20)
```

Resultado:

```
La suma es: 8
La suma es: 30
```

Orden importa

Los argumentos se asignan a los parámetros en el mismo orden.

```
def presentar(nombre, edad):
    print("Nombre:", nombre)
    print("Edad:", edad)

presentar("Luis", 20)
```

Resultado:

```
Nombre: Luis
Edad: 20
```

Si se llama `presentar(20, "Luis")`, los valores quedan invertidos.

Return

Hasta ahora las funciones solo imprimen resultados.

Con `return`, una función puede devolver un valor que el programa puede usar.

Sin return

```
def sumar(a, b):
    print(a + b)

sumar(3, 5)  # imprime 8 pero no guarda nada
```

Con return

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

resultado = sumar(3, 5)
print("El resultado es:", resultado)
```

Resultado:

```
El resultado es: 8
```

- `return` devuelve el valor al lugar donde se llamó la función.
- El valor puede guardarse en una variable para usarlo después.

¿Por qué return es mejor que print?

| Característica | print dentro | return | | --- | --- | --- | | ¿Se puede guardar el resultado? | No | Sí | | ¿Se puede usar en otra operación? | No | Sí | | ¿Se puede imprimir después? | No aplica | Sí |

Ejemplo con return

```
def calcular_area(base, altura):
    return base * altura

area = calcular_area(5, 3)
print("El área es:", area)
print("El doble del área es:", area * 2)
```

Resultado:

```
El área es: 15
El doble del área es: 30
```

Return termina la función

Cuando Python encuentra `return`, la función termina de inmediato.

El código después de `return` no se ejecuta.

Ejemplo

```
def verificar(numero):
    if numero < 0:
        return "El número es negativo"
    return "El número es positivo o cero"

print(verificar(-5))
print(verificar(10))
```

Resultado:

```
El número es negativo
El número es positivo o cero
```

- Si `numero` es negativo, el primer `return` termina la función.
- El segundo `return` solo se alcanza si el número no es negativo.

Ejemplo completo 1: Área de rectángulo

```
def area_rectangulo(base, altura):
    return base * altura

a1 = area_rectangulo(5, 3)
a2 = area_rectangulo(8, 2)
a3 = area_rectangulo(10, 4)

print("Área 1:", a1)
print("Área 2:", a2)
print("Área 3:", a3)
```

Resultado:

```
Área 1: 15
Área 2: 16
Área 3: 40
```

La fórmula está en un solo lugar. Si cambia, se modifica solo ahí.

Ejemplo completo 2: Calculadora

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

def restar(a, b):
    return a - b

def multiplicar(a, b):
```

```
    return a * b

x = float(input("Primer número: "))
y = float(input("Segundo número: "))

print("Suma:", sumar(x, y))
print("Resta:", restar(x, y))
print("Multiplicación:", multiplicar(x, y))
```

Ventaja

Cada operación está en su propia función.

El programa es más fácil de leer y modificar.

Ejemplo completo 3: Función con ciclo

```
def tabla_multiplicar(numero):
    for i in range(1, 11):
        print(numero, "x", i, "=", numero * i)

tabla_multiplicar(3)
print()
tabla_multiplicar(7)
```

Resultado parcial:

```
3 x 1 = 3
3 x 2 = 6
...
3 x 10 = 30

7 x 1 = 7
...
7 x 10 = 70
```

Las funciones pueden contener cualquier estructura: `if`, `while`, `for`.

Errores comunes

Error 1: llamar antes de definir

```
saludar()  # ERROR: la función no existe todavía

def saludar():
    print("Hola")
```

La función debe definirse antes de llamarla.

Error 2: olvidar los paréntesis

```
def saludar():
    print("Hola")

saludar   # no hace nada — solo referencia la función
saludar() # correcto — ejecuta la función
```

Error 3: número incorrecto de argumentos

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

sumar(5)      # ERROR: falta un argumento
sumar(5, 3, 2)  # ERROR: demasiados argumentos
```

La cantidad de argumentos debe coincidir con la cantidad de parámetros.

Error 4: olvidar return

```
def multiplicar(a, b):
    a * b  # calcula pero no devuelve nada

resultado = multiplicar(3, 4)
print(resultado)  # imprime None
```

Sin `return`, la función devuelve `None` automáticamente.

Error 5: usar el resultado de una función sin return

```
def mostrar_doble(x):
    print(x * 2)

resultado = mostrar_doble(5)  # imprime 10
print(resultado)              # imprime None
```

`print` muestra el valor en pantalla pero no lo devuelve.

Variables locales

Las variables creadas dentro de una función solo existen dentro de ella.

No se pueden usar fuera.

Ejemplo

```
def calcular():
    resultado = 100
    print("Dentro:", resultado)

calcular()
print("Fuera:", resultado)  # ERROR: resultado no existe aquí
```

¿Por qué?

Cada función tiene su propio espacio de variables.

Esto evita que funciones distintas se interfieran entre sí.

Buenas prácticas

- Dar nombres descriptivos a las funciones: `calcular_promedio` es mejor que `f`.
- Una función debe hacer una sola cosa y hacerla bien.
- Si una función tiene `return`, usarla para devolver el resultado — no mezclar `return` y `print` innecesariamente.
- Definir todas las funciones al inicio del programa, antes de llamarlas.
- Mantener las funciones cortas: si una función tiene más de 20 líneas, probablemente hace demasiado.