

# Programación con



## Tema 5: Funciones

*Alma Mallo Casdelo - [alma.mallo@udc.es](mailto:alma.mallo@udc.es)*

# Funciones

---

- Una función es un conjunto de sentencias que se agrupan en una entidad para realizar una tarea determinada fácilmente siempre que sea necesario.
- Las funciones evitan repetir sentencias y facilitan el desarrollo (divide y vencerás).
- Definir una función implica darle un nombre, especificar sus argumentos y, finalmente, las sentencias que la forman.

# Definición

---

```
def pregunta_edad():  
    """Pregunta la edad al usuario y muestra un mensaje acorde por pantalla."""  
    edad = int(input("¿Cuántos años tienes? "))  
    if edad < 20:  
        print("Eres un chavalín.")  
    elif edad < 40:  
        print("¡Estás en lo mejor!")  
    elif edad < 60:  
        print("Cuidado con las lesiones.")  
    else:  
        print("Ten cuidado, ya no estás en garantía.")
```

} Definición de la función

```
if __name__ == "__main__":  
    pregunta_edad()
```

→ Ejecución de la función

# Definición

---

```
def pregunta_edad():  
    """Pregunta la edad al usuario y muestra un mensaje acorde por pantalla."""  
    edad = int(input("¿Cuántos años tienes? "))  
    if edad < 20:  
        print("Eres un chavalín.")  
    elif edad < 40:  
        print("¡Estás en lo mejor!")  
    elif edad < 60:  
        print("Cuidado con las lesiones.")  
    else:  
        print("Ten cuidado, ya no estás en garantía.")  
  
if __name__ == "__main__":  
    pregunta_edad()
```

Comentario explicativo de la función: doctring

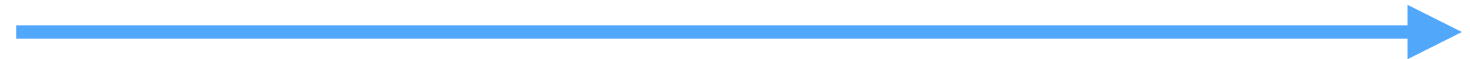
Sentencias de la función. Están indentadas hacia la derecha.

Esto hace que la función solo se ejecute si se solicita al intérprete la ejecución de este fichero. Si se hace un import, no se ejecuta la función.

# Devolución de resultados

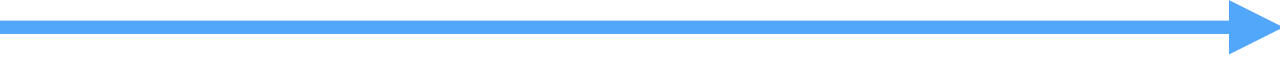
---

```
def pregunta_edad():  
    """Pregunta la edad al usuario."""  
    edad = int(input("¿Cuántos años tienes? "))  
    return edad
```



Las funciones pueden devolver resultados...

```
def main():  
    """Función principal del programa."""  
    edad = pregunta_edad()  
    if edad < 20:  
        print("Eres un chavalín.")  
    elif edad < 40:  
        print("¡Estás en lo mejor!")  
    elif edad < 60:  
        print("Cuidado con las lesiones.")  
    else:  
        print("Ten cuidado, ya no estás en garantía.")
```



... que se recogen de esta forma.

```
if __name__ == "__main__":  
    main()
```

# Variables locales

---

```
def pregunta_edad():  
    """Pregunta la edad al usuario."""  
    edad = int(input("¿Cuántos años tienes? "))  
    return edad
```

Las variables pueden llamarse igual...

```
def main():  
    """Función principal del programa."""  
    edad = pregunta_edad()  
    if edad < 20:  
        print("Eres un chavalín.")  
    elif edad < 40:  
        print("¡Estás en lo mejor!")  
    elif edad < 60:  
        print("Cuidado con las lesiones.")  
    else:  
        print("Ten cuidado, ya no estás en garantía.")
```

```
if __name__ == "__main__":  
    main()
```

# Variables locales

---

```
def pregunta_edad():  
    """Pregunta la edad al usuario."""  
    respuesta = int(input("¿Cuántos años tienes? "))  
    return respuesta  
  
def main():  
    """Función principal del programa."""  
    edad = pregunta_edad()  
    if edad < 20:  
        print("Eres un chavalín.")  
    elif edad < 40:  
        print("¡Estás en lo mejor!")  
    elif edad < 60:  
        print("Cuidado con las lesiones.")  
    else:  
        print("Ten cuidado, ya no estás en garantía.")  
  
if __name__ == "__main__":  
    main()
```

... o no, da igual.

Las variables “definidas” en una función son locales, solo accesibles dentro de esa misma función.  
Si aparece en otra función... simplemente es otra variable con el mismo nombre.

# Variables globales

---

- Si a una variable se le asigna un valor fuera de cualquier función, dicha variable es global.
- Su uso está desaconsejado por ser fuente de errores.
- En Python las funciones pueden usar una variable global simplemente accediendo a ella... pero para modificarla tienen que indicar explícitamente que dicha variable es global (en realidad, esto funciona como recordatorio para el programador).



# Variables globales

---

```
def inc_v1(n):  
    n = x + 1  
    return n
```

→ Esta variable "x" SÍ es la variable global.

```
def inc_v2(n):  
    x = n + 1  
    return x
```

→ Esta variable "x" NO es la variable global, si no una variable local con el mismo nombre.

```
def inc_v3(n):  
    global x  
    x = n + 1  
    return x
```

→ Esta variable "x" SÍ es la variable global.

```
x = 2  
print(x)  
print(inc_v1(x))  
print(x)  
print(inc_v2(x))  
print(x)  
print(inc_v3(x))  
print(x)
```

Salida por pantalla:

2  
3  
2  
3  
2  
3  
3

# Paso de argumentos

```
def pregunta_nombre():  
    """Pregunta el nombre al usuario."""  
    return input("¿Cómo te llamas? ")  
  
def pregunta_edad(nombre):  
    """Pregunta la edad al usuario."""  
    edad = int(input(f"¡Hola {nombre}!. ¿Cuántos años tienes? "))  
    return edad  
  
def evalua_edad(edad):  
    """Muestra un mensaje acorde a la edad del usuario."""  
    if edad < 20:  
        print("Eres un chavalín.")  
    elif edad < 40:  
        print("¡Estás en lo mejor!")  
    elif edad < 60:  
        print("Cuidado con las lesiones.")  
    else:  
        print("Ten cuidado, ya no estás en garantía.")  
  
def main():  
    """Función principal del programa."""  
    nombre = pregunta_nombre()  
    edad = pregunta_edad(nombre)  
    evalua_edad(edad)  
  
if __name__ == "__main__":  
    main()
```

A la derecha de “return” no tiene por qué ir el nombre de una variable, puede ir cualquier expresión que, al ser evaluada, produzca un dato correcto (en el sentido de que el tipo de dato es el esperado).

Las funciones pueden tener argumentos (datos que necesitan para hacer lo que tienen que hacer) que se ponen entre los paréntesis tanto al definir la función como al utilizarla:

- No se pueden omitir nunca los paréntesis, ni al definirla ni al ejecutarla, aunque no haya argumentos.
- Los argumentos son variables locales, simplemente están inicializados a un valor establecido en tiempo de ejecución en el momento de ser llamada la función.
- Como con cualquier variable local, los nombres NO tienen por qué coincidir en la definición de la función y en la llamada a la función, aunque a veces lo hagan. Son variables diferentes.
- En Python, las variables son todas punteros a objetos, así que una función puede modificar los objetos subyacentes que se pasan como argumentos dependiendo de cómo se usen.

# Argumentos y *return* por posición

```
def entrada_datos():  
    """Pide por teclado base y exponente."""  
    base = int(input("Introduce la base: "))  
    exponente = int(input("Introduce el exponente (>0): "))  
    while exponente <= 0:  
        exponente = int(input("Error en el rango. Introduce el número (>0): "))  
    return base, exponente
```

Una función puede devolver más de un dato separando las expresiones por comas.

```
def calc_potencia(base, exponente):  
    """Calcula la potencia."""  
    potencia = 1  
    for i in range(exponente):  
        potencia *= base  
    return potencia
```

Una función puede tener más de un argumento separándolos por comas.

```
def imprime_salida(base, exponente, potencia):  
    """Muestra el resultado por pantalla."""  
    print(f"{base} elevado a {exponente} vale {potencia}")
```

```
def main():  
    """Función principal del programa."""  
    base, exponente = entrada_datos()  
    potencia = calc_potencia(base, exponente)  
    imprime_salida(base, exponente, potencia)
```

Cuando una función devuelve más de un dato se recogen EN EL MISMO ORDEN separando por comas también.

```
if __name__ == "__main__":  
    main()
```

Cuando una función tiene más de un argumento, se pasan EN EL MISMO ORDEN separando por comas también.

# Argumentos por nombre

---

```
def entrada_datos():  
    """Pide por teclado base y exponente."""  
    base = int(input("Introduce la base: "))  
    exponente = int(input("Introduce el exponente (>0): "))  
    while exponente <= 0:  
        exponente = int(input("Error en el rango. Introduce el número (>0): "))  
    return base, exponente
```

```
def calc_potencia(base, exponente):  
    """Calcula la potencia."""  
    potencia = 1  
    for i in range(exponente):  
        potencia *= base  
    return potencia
```

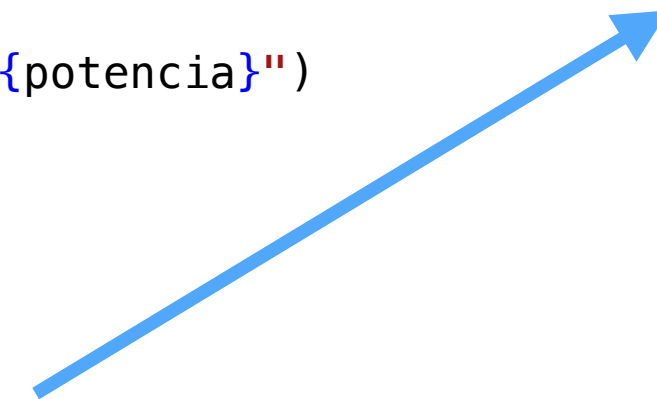
```
def imprime_salida(base, exponente, potencia):  
    """Muestra el resultado por pantalla."""  
    print(f"{base} elevado a {exponente} vale {potencia}")
```

```
def main():  
    """Función principal del programa."""  
    base, exponente = entrada_datos()  
    potencia = calc_potencia(exponente=exponente, base=base)  
    imprime_salida(base, exponente, potencia)
```

```
if __name__ == "__main__":  
    main()
```

Los argumentos pueden indicarse también mediante una asignación al nombre utilizado en la definición de la función. Implicaciones:

- Es un poco más tedioso llamar a la función pero...
- Facilita utilizar funciones con muchos argumentos porque no es necesario ponerlos todos al llamar a la función y sobre todo...
- Disminuye la posibilidad de cometer errores al no importar el orden en el que se ponen los argumentos.



# Argumentos con valor por defecto

---

```
def calc_potencia(base, exponente=2):  
    """Calcula la potencia."""  
    potencia = 1  
    for i in range(exponente):  
        potencia *= base  
    return potencia
```

```
def main():  
    """Función principal del programa."""  
    n = int(input("Dime un número entero: "))  
    cuadrado = calc_potencia(n)  
    print(n, "al cuadrado vale", cuadrado)
```

```
if __name__ == "__main__":  
    main()
```

- Se pueden mezclar argumentos por posición y argumentos por nombre a la hora de llamar a una función pero, en ese caso, primero van los que se especifican por posición y después los que se especifican por nombre.
- Cuando se define una función, los argumentos pueden tomar un valor por defecto de forma que, cuando se llama a la función, si no se especifica ningún valor para ellos, toman dicho valor por defecto.

# Lista variable de argumentos

---

- Se pueden hacer funciones con un número variable de argumentos:
- Si un argumento tiene **un asterisco** delante de su nombre, este será **una tupla** que contendrá todos los argumentos por posición que no hayan aparecido explícitamente antes.
- Si un argumento tiene **dos asteriscos** delante de su nombre, este será **un diccionario** que contendrá todos los argumentos por nombre que no hayan aparecido explícitamente antes.



# Lista variable de argumentos

---

```
def normaliza(texto, *args, izq=0, der=1, **kwargs):
    """Normaliza números en un rango determinado."""
    if kwargs:
        print("Advertencia: he recibido argumentos desconocidos: ", kwargs)
    print(texto, end=" ")
    minimo = min(args)
    maximo = max(args)
    constante = (der - izq) / (maximo - minimo)
    for num in args:
        print((num - minimo) * constante + izq, end=" ")
    print()

if __name__ == "__main__":
    normaliza("Resultado de normalizar:", 1, 2, 3, 4, 5, 6, izq=1, der=2, no_valgo_para_nada=11)
```

Salida por pantalla:

Advertencia: he recibido argumentos desconocidos: {'no\_valgo\_para\_nada': 11}  
Resultado de normalizar: 1.0 1.2 1.4 1.6 1.8 2.0

# Funciones incorporadas

---

## Built-in Functions

### A

**abs()**  
**aiter()**  
**all()**  
**any()**  
**anext()**  
**ascii()**

### B

**bin()**  
**bool()**  
**breakpoint()**  
**bytearray()**  
**bytes()**

### C

**callable()**  
**chr()**  
**classmethod()**  
**compile()**  
**complex()**

### D

**delattr()**  
**dict()**  
**dir()**  
**divmod()**

### E

**enumerate()**  
**eval()**  
**exec()**

### F

**filter()**  
**float()**  
**format()**  
**frozenset()**

### G

**getattr()**  
**globals()**

### H

**hasattr()**  
**hash()**  
**help()**  
**hex()**

### I

**id()**  
**input()**  
**int()**  
**instance()**  
**issubclass()**  
**iter()**

### L

**len()**  
**list()**  
**locals()**

### M

**map()**  
**max()**  
**memoryview()**  
**min()**

### N

**next()**

### O

**object()**  
**oct()**  
**open()**  
**ord()**

### P

**pow()**  
**print()**  
**property()**

### R

**range()**  
**repr()**  
**reversed()**  
**round()**

### S

**set()**  
**setattr()**  
**slice()**  
**sorted()**  
**staticmethod()**  
**str()**  
**sum()**  
**super()**

### T

**tuple()**  
**type()**

### V

**vars()**

### Z

**zip()**

**\_\_import\_\_()**



# Funciones incorporadas

---

- E/S:
  - `input(p)` => muestra `p` y lee una línea por teclado
  - `print(*o)` => muestra por pantalla
- Devuelven un iterador:
  - `range(b, e, s)` => itera en intervalo `[b:e]` de `s` en `s`
  - `enumerate(i)` => itera utilizando posición y valor
  - `zip(*i)` => iteran en paralelo sobre varios iterables
  - `map(f, i)` => aplica `f` a elementos de un iterable
  - `filter(f, i)` => elem. de `i` para los que `f` es `True`
- `len(s)` => número de elementos
- Matemáticas:
  - `abs(x)` => valor absoluto
  - `round(n, d)` => redondea `n` con `d` dígitos
  - `max(i)` => máximo de los elementos de un iterable
  - `min(i)` => mínimo de los elementos de un iterable
  - `sum(i)` => suma de los elementos de un iterable
- Conversión de tipos:
  - `int(x)` => convierte a `int`
  - `float(x)` => convierte a `float`
- ...

# Funciones lambda

---

- Inspiradas en los lenguajes de programación funcionales.
- En Python, son expresiones que devuelven funciones anónimas:
  - Constan únicamente de una expresión (no vale otro tipo de sentencia) que se aplica a los argumentos.
  - No tienen nombre... aunque se pueden asignar (recuerda que todo son objetos, incluidas las funciones), con lo que, en la práctica, sí pueden tenerlo. Ejemplo con dos argumentos:

```
suma = lambda x, y: x + y
```

- Función equivalente:

```
def suma(x, y):  
    return x + y
```

- Se usan, sobre todo, como argumento de funciones de orden superior (funciones que tienen a otras funciones como argumentos) como `map()` o `filter()`. Ejemplo:

```
lista = [10, 25, 17, 9, 30, -5]  
lista_cuadrado = map(lambda n: n ** 2, lista)
```

# Decoradores

---

- Los decoradores son funciones de orden superior que:
  - Su objetivo es modificar / extender el comportamiento de una función que se le pasa como argumento.
  - Devuelve dicha función modificada (=> “decorada”).
- Hay varios definidos en las librerías estándar, y hay muchas librerías que también definen sus propios decoradores.

# Decoradores

---

```
def hola(funcion):  
    """Decora una función anteponiendo 'Hola'."""  
  
    def funcion_decorada(texto):  
        print("Hola", end=" ")  
        funcion(texto)  
  
    return funcion_decorada
```

```
@hola  
def muestra_nombre(nombre):  
    """Muestra el nombre por pantalla."""  
    print(nombre)
```

```
if __name__ == "__main__":  
    muestra_nombre("Jose")
```

.....

Equivalencia SIN usar el decorador @hola:

```
if __name__ == "__main__":  
    funcion_decorada = hola(muestra_nombre)  
    funcion_decorada("Jose")
```

.....

# Recursividad

---

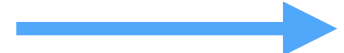
```
def entrada_datos():  
    """Pide por teclado base y exponente."""  
    base = int(input("Introduce la base: "))  
    exponente = int(input("Introduce el exponente (>0): "))  
    while exponente <= 0:  
        exponente = int(input("Error en el rango. Introduce el número (>0): "))  
    return base, exponente
```

```
def calc_potencia(base, exponente):  
    """Calcula la potencia."""  
    if exponente == 0:  
        return 1  
    return base * calc_potencia(base, exponente - 1)
```

```
def imprime_salida(base, exponente, potencia):  
    """Muestra el resultado por pantalla."""  
    print(f"{base} elevado a {exponente} vale {potencia}")
```

```
def main():  
    """Función principal del programa."""  
    base, exponente = entrada_datos()  
    potencia = calc_potencia(base, exponente)  
    imprime_salida(base, exponente, potencia)
```

```
if __name__ == "__main__":  
    main()
```

- 
- Una función se puede llamar a sí misma (función recursiva). En ese caso:
    - Es necesario que exista al menos un caso en el que la función no se llame a sí misma (caso base), o tendremos un desbordamiento de pila.
  - La recursividad permite implementar algunas definiciones matemáticas / operaciones que por su propia naturaleza son recursivas de forma:
    - Más concisa.
    - Pero menos eficiente.

# Referencias

---

- [https://docs.python.org/3/reference/compound\\_stmts.html#function-definitions](https://docs.python.org/3/reference/compound_stmts.html#function-definitions)
- <https://docs.python.org/3/library/functions.html>