

# U3.5

# Funciones



# Indice

# Funciones en Python

- Las funciones son estructuras esenciales de código.
- Unidades lógicas del programa.
- Dividen y organizan el código para mayor legibilidad y reutilización.
- Agrupan instrucciones para resolver un problema concreto.
- Objetivos:
  - Dividir y organizar el código.
  - Encapsular código repetitivo.

# Cómo definir una función en Python

- Utiliza la palabra reservada `def`.
- Nombre de la función seguido de paréntesis y lista de parámetros (opcional).
- Cabecera termina con dos puntos.
- Cuerpo de la función con sangrado mayor.
- Opcional: instrucción `return` para devolver un resultado.
- docstring se utiliza para documentar la función.

# Cómo usar una función en Python

- Para invocarla, escribe su nombre como una instrucción.
- Pasa los argumentos según los parámetros definidos.

python [Copy code](#)

```
def multiplica_por_5(numero):  
    print(f'{numero} * 5 = {numero * 5}')    # 7 * 5 = 35  
  
multiplica_por_5(7)
```

# Parámetros y argumentos

# Diferencia entre parámetro y argumento

- Parámetro: definido en la función.
- Argumento: valor pasado a la función al ser invocada.

python [Copy code](#)

```
def multiplica_por_5(numero): # 'numero' es un parámetro
    print(f'{numero} * 5 = {numero * 5}')
```

```
multiplica_por_5(7) # '7' es un argumento
```



# Parámetros

- Paso por valor: copia el valor de las variables.
- Paso por referencia: copia la dirección de memoria.

# `*args` y `**kwargs` en Python

# Significado de `*args` y `**kwargs` en Python

- Permiten funciones con un número variable de argumentos.
- Proporcionan flexibilidad en la cantidad y tipo de argumentos.

# Uso de \*args

- `*args` permite argumentos sin palabras clave.

python

 Copy code

```
def sumar(*args):  
    return sum(args) # sum es una función incorporada de Python  
  
print(sumar(3, 5, 10, 15)) # Imprime 33
```

# Uso de `**kwargs`

- `**kwargs` permite argumentos con palabras clave.

python [Copy code](#)

```
def describir_persona(**kwargs):
    for clave, valor in kwargs.items():
        print(f"{clave}: {valor}")

describir_persona(nombre="John", edad=25, ciudad="Nueva York")
```

Note: `**kwargs` permite pasar un diccionario variable de argumentos con palabras clave.

## Ejemplo de `*args` para sumar números

python [Copy code](#)

```
def sum(*args):
    value = 0
    for n in args:
        value += n
    return value
```

Note: Ejemplos de funciones que utilizan `*args`.

# Ejemplo de `**kwargs` para filtrar datos

python

 Copy code

```
def filtrar(**kwargs):  
    condiciones = " AND ".join([f"{k}='{v}'" for k, v in kwargs.items()])  
    return f"SELECT * FROM clientes WHERE {condiciones};"
```

# El Orden Importa

- En la definición de la función, usa `*args` y luego `**kwargs`.


python

 Copy code

```
def ejemplo(arg1, arg2, *args, **kwargs):  
    pass
```

# Usando \*args para Desempaquetar una Lista o Tupla

python


 Copy code

```
def resultado(x, y, op):  
    if op == '+':  
        return x + y  
    elif op == '-':  
        return x - y  
  
a = (1, 2, '+')  
print(resultado(*a))    # Imprime 3
```



# Usando `**kwargs` para Desempaquetar un Diccionario

python

 Copy code

```
a = {"op": "+", "x": 2, "y": 5}
print(resultado(**a)) # Imprime 7
```

# Parámetros Opcionales en Funciones Python

# ¿Qué son los parámetros opcionales?

- Los parámetros opcionales en una función tienen valores predeterminados.
- Toman esos valores si no se proporciona un valor específico al invocar la función.

# Ejemplo de Función con Parámetro Opcional

python  Copy code

```
def saludo(nombre, mensaje="encantado de saludarte") :  
    print("Hola {}, {}".format(nombre, mensaje))
```

- El parámetro `nombre` es obligatorio.
- El parámetro `mensaje` es opcional y tiene un valor predeterminado.

# Uso de la Función con Parámetro Opcional

python  Copy code

```
saludo("Juan")  
# Salida: Hola Juan, encantado de saludarte  
  
saludo("Ana", "bienvenida")  
# Salida: Hola Ana, bienvenida
```

- En la primera llamada, se usa el valor predeterminado para mensaje.
- En la segunda llamada, se proporciona un valor específico para mensaje.

# Restricciones en la Definición de Parámetros Opcionales

- Una vez definido un parámetro opcional, todos los parámetros a su derecha también deben ser opcionales.

python  Copy code

```
# Incorrecto
def ejemplo(param1, param2="opcional", param3):
    # Código de la función

# Correcto
def ejemplo(param1, param2="opcional", param3="otro opcional"):
    # Código de la función
```

# Beneficios de Parámetros Opcionales

- **Flexibilidad:** Permite a los usuarios de la función adaptarla según sus necesidades.
- **Valores Predeterminados:** Proporciona valores por defecto para evitar errores en llamadas incompletas.
- **Facilita el Uso:** Simplifica el uso de funciones al reducir la necesidad de proporcionar todos los argumentos.

# Retorno de valores



# Sentencia return

- El retorno de valores es opcional, puede devolver o no un valor.
- Termina la ejecución de la función y continúa el programa.

python [Copy code](#)

```
def cuadrado_de_par(numero):  
    if not numero % 2 == 0:  
        return  
    else:  
        print(numero ** 2)    # 64
```

```
cuadrado_de_par(8)
```

# Devolver más de un valor

- En Python, se puede devolver más de un valor con `return`.
- En tal caso, por defecto, se devuelve una tupla de valores.

python [Copy code](#)

```
def cuadrado_y_cubo(numero):  
    return numero ** 2, numero ** 3
```

```
cuad, cubo = cuadrado_y_cubo(4)  
cuad    # 16  
cubo    # 64
```

# Devolver resultados en una lista

- Se pueden devolver diferentes resultados en una lista.

python [Copy code](#)

```
def tabla_del(numero):  
    resultados = []  
    for i in range(11):  
        resultados.append(numero * i)  
    return resultados  
  
res = tabla_del(3)  
res # [0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

# Python siempre devuelve un valor

- A diferencia de otros lenguajes, Python no tiene procedimientos.
- Si no hay `return`, se devuelve automáticamente `None`.

python [Copy code](#)

```
def saludo(nombre):  
    print(f'Hola {nombre}')
```

```
print(saludo('j2logo')) # Hola j2logo \n None
```

# Variables y ámbito

# Ámbito y ciclo de vida de las variables

- Local: dentro de una función, no accesible fuera.
- Global: definidas fuera de funciones, visibles dentro.

python [Copy code](#)

```
def muestra_x():  
    x = 10  
    print(f'x vale {x}')    # x vale 10  
  
x = 20  
muestra_x()  
print(x)    # 20
```

# Ámbito y ciclo de vida de las variables

- Ámbito local: dentro de una función.
- Variables desaparecen al finalizar la función.

# Variables globales

- Ámbito global: fuera de funciones.
- Pueden ser consultadas dentro de funciones.

python [Copy code](#)

```
y = 20
def muestra_x():
    global x
    x += 1
    print(f'x vale {x}')
    print(f'y vale {y}')
```



# Funciones Lambda en Python

- Funciones anónimas y pequeñas.
- No requieren definición con `def`.
- **Sintaxis:** `lambda argumentos: expresion.`

python [Copy code](#)

```
doble = lambda x: x * 2
```

# Resumen I

- Funciones en Python son unidades lógicas de código.
- Dividen y organizan el código, facilitando la reutilización.
- Se definen con `def`, seguido del nombre y parámetros (opcional).
- Se invocan escribiendo el nombre y pasando argumentos.
- `*args` para argumentos no clave.
- `*args` recoge en una tupla.

# Resumen II

- `**kwargs` para argumentos clave.
- `**kwargs` recoge en un diccionario.
- Orden en la definición: `*args` primero, luego `**kwargs`.
- `return` opcional para devolver resultados.
- Ámbito de las variables: local y global.
- Las funciones lamda son utiles para casos simples y breves.

¡Gracias por su atención!