Python
Funciones

CertiDevs

Índice de contenidos

1. Introducción

Las **funciones** en Python son bloques de código reutilizables que realizan una tarea específica.

Pueden recibir **argumentos** y devolver **valores**.

Las funciones permiten organizar y modularizar el código, lo que facilita la comprensión, el mantenimiento y la depuración del programa.

2. Definición de funciones

Para definir una **función** en Python, se utiliza la palabra clave `def`, seguida del **nombre de la función**, **paréntesis** y **dos puntos** (`:`).

El **cuerpo** de la función debe estar **indentado**.

Aquí hay un ejemplo de una función simple llamada `saludar` que imprime un mensaje de bienvenida:

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

3. Llamada a funciones

Para **llamar** o **invocar** a una función en Python, simplemente escribe el **nombre de la función seguido de paréntesis**.

Ejemplo de llamada a la función saludar:

```
saludar()
```

4. Argumentos de funciones

Las funciones en Python pueden recibir **argumentos**, que son valores que se pasan a la función cuando se llama.

Los argumentos se especifican **entre paréntesis** después del nombre de la función en la definición de la función.

Aquí hay un ejemplo de una función llamada `saludar_persona` que recibe un argumento llamado nombre:

```
def saludar_persona(nombre):
    print(f"¡Hola, {nombre}!")
```

Llamando a la función saludar_persona con un argumento:

```
saludar_persona("Juan")
```

5. Valores de retorno

Las funciones en Python pueden **devolver valores** utilizando la palabra clave `return`.

Cuando se ejecuta una declaración `return`, la función termina y devuelve el valor especificado.

Aquí hay un ejemplo de una función llamada suma que devuelve la suma de dos números:

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

Llamando a la función suma y almacenando el valor devuelto en una variable:

```
resultado = suma(3, 5)
print(resultado)
```

6. Argumentos predeterminados

En Python, puedes proporcionar **valores predeterminados para los argumentos** de una función.

Esto permite llamar a la función sin proporcionar explícitamente esos argumentos, en cuyo caso se utilizarán los valores predeterminados.

Aquí hay un ejemplo de una función llamada `potencia` que calcula la potencia de un número.

El argumento `exponente` tiene un valor predeterminado de `2`:

```
def potencia(base, exponente=2):
    return base ** exponente
```

Llamando a la función potencia sin proporcionar el argumento exponente:

```
cuadrado = potencia(4)
print(cuadrado)
```

7. Argumentos de palabras clave

En Python, puedes llamar a una función con argumentos utilizando **palabras clave**.

Esto permite especificar explícitamente qué argumento recibe qué valor y puede mejorar la legibilidad del código.

Ejemplo de llamada a la función `potencia` utilizando argumentos de **palabras clave**:

```
resultado = potencia(base=3, exponente=4)
print(resultado)
```

8. Argumentos de longitud variable

Las funciones en Python pueden recibir un número variable de argumentos utilizando el operador de desempaquetado `*` para argumentos posicionales y `**` para argumentos de palabras clave.

Aquí hay un ejemplo de una función llamada `suma_multiples` que suma un número variable de argumentos:

```
def suma_multiples(*numeros):
    total = 0
    for num in numeros:
        total += num
    return total
```

Llamando a la función suma_multiples con diferentes cantidades de argumentos:

```
print(suma_multiples(1, 2, 3))
print(suma_multiples(4, 5))
```

9. Funciones anónimas (lambda)

Python permite crear **funciones anónimas**, también llamadas funciones **lambda**, utilizando la palabra clave `lambda`.

Las **funciones lambda** son funciones pequeñas y simples que se pueden definir en una sola línea de código.

Aquí hay un ejemplo de una **función lambda** que suma dos números:

```
suma = lambda a, b: a + b
```

Llamando a la función lambda suma:

```
resultado = suma(3, 5)
print(resultado)
```

A diferencia de las funciones normales que se definen con la palabra clave def, las funciones lambda se definen utilizando la palabra clave lambda.

Las funciones lambda son útiles en situaciones donde necesitas una función simple que puede ser definida de forma breve e inline, como al pasar una función a otra función, en lugar de definir una función completa con def.

A continuación, se presentan varios ejemplos útiles de funciones lambda en diferentes escenarios:

9.1. Ejemplo 1: Ordenar una lista de tuplas por el segundo elemento

Supongamos que tienes una lista de tuplas y deseas ordenar la lista por el segundo elemento de cada tupla. Puedes usar la función sorted con una función lambda como argumento para la clave de ordenamiento:

```
lista_de_tuplas = [(1, 5), (3, 1), (4, 9), (2, 7)]
lista_ordenada = sorted(lista_de_tuplas, key=lambda tupla: tupla[1])
print(lista_ordenada)  # Output: [(3, 1), (1, 5), (2, 7), (4, 9)]
```

9.2. Ejemplo 2: Aplicar una función a todos los elementos de una lista

Supongamos que tienes una lista de números y deseas elevar al cuadrado cada número de la lista. Puedes usar la función map con una función lambda para aplicar la operación a cada elemento:

```
numeros = [1, 2, 3, 4, 5]
cuadrados = list(map(lambda x: x**2, numeros))
print(cuadrados)  # Output: [1, 4, 9, 16, 25]
```

9.3. Ejemplo 3: Filtrar elementos de una lista

Supongamos que tienes una lista de números y deseas conservar solo los números pares. Puedes usar la función filter con una función lambda para filtrar los elementos según la condición:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # Output: [2, 4, 6, 8]
```

9.4. Ejemplo 4: Calcular el producto de una lista de números

Supongamos que tienes una lista de números y deseas calcular el producto de todos los elementos de la lista. Puedes usar la función reduce del módulo functools con una función lambda para acumular el resultado:

```
from functools import reduce

numeros = [1, 2, 3, 4, 5]
producto = reduce(lambda x, y: x * y, numeros)
print(producto)  # Output: 120
```

9.5. Ejemplo 5: Combinar dos listas en un diccionario

Supongamos que tienes dos listas, una con claves y otra con valores, y deseas combinarlas en un diccionario. Puedes usar la función zip junto con una comprensión de diccionarios y una función lambda para crear el diccionario:

```
claves = ['a', 'b', 'c', 'd']
valores = [1, 2, 3, 4]

diccionario = {k: v for k, v in zip(claves, valores)}
print(diccionario) # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

En este caso, no fue necesario usar una función lambda porque la comprensión de diccionarios es suficiente para combinar las dos listas. Sin embargo, aquí hay un ejemplo alternativo utilizando `map` y una función lambda para lograr el mismo resultado:

```
claves = ['a', 'b', 'c', 'd']
valores = [1, 2, 3, 4]

diccionario = dict(map(lambda k, v: (k, v), claves, valores))
print(diccionario)  # Output: {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

10. Decoradores de funciones

Los **decoradores** son una herramienta en Python que permite modificar o mejorar el comportamiento de una función sin cambiar su código fuente.

Un decorador es una función que toma otra función como argumento y devuelve una nueva función que generalmente extiende o modifica el comportamiento de la función original.

Aquí hay un ejemplo simple de un decorador que registra cuándo se llama a una función:

```
def registrar_llamada(func):
    def envoltorio(*args, **kwargs):
        print(f"Se llamó a la función: {func.__name__}")
        return func(*args, **kwargs)
    return envoltorio

@registrar_llamada
def suma(a, b):
    return a + b

resultado = suma(3, 5)
```

11. Funciones de orden superior

Las **funciones de orden superior** son funciones que toman una o más **funciones como argumentos** y/o devuelven una función como resultado.

Esto es posible en Python porque las funciones son **objetos** de primera clase, lo que significa que pueden ser tratadas como valores y pasadas como argumentos a otras funciones.

Ejemplo de función de orden superior `aplicar_operacion` que toma una función de operación y dos números como argumentos:

```
def aplicar_operacion(func, a, b):
    return func(a, b)

def suma(a, b):
    return a + b

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

resultado1 = aplicar_operacion(suma, 3, 5)
resultado2 = aplicar_operacion(resta, 10, 4)
print(resultado1, resultado2)
```

12. Funciones parcialmente aplicadas

Las **funciones parcialmente aplicadas** son funciones que tienen algunos de sus argumentos "preestablecidos" o "fijos".

Esto se puede lograr en Python utilizando la función partial del módulo `functools`.

Ejemplo de función parcialmente aplicada `suma_cinco` que suma 5 a su único argumento:

```
from functools import partial
```

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

suma_cinco = partial(suma, 5)

resultado = suma_cinco(3)
print(resultado)
```

13. Funciones de cierre (closures)

Un **cierre** es una función que "recuerda" el entorno en el que fue creada, incluso si ese entorno ya no existe.

Esto permite que una función "capture" valores de su entorno circundante y los use en llamadas futuras.

Ejemplo de función de cierre `crear_multiplicador` que crea una función que multiplica su argumento por un factor específico:

```
def crear_multiplicador(factor):
    def multiplicador(n):
        return n * factor
    return multiplicador

duplicar = crear_multiplicador(2)
triplicar = crear_multiplicador(3)

print(duplicar(4))
print(triplicar(4))
```

En este ejemplo, la función multiplicador es un cierre que captura el valor de factor del entorno en el que fue creada.