**Funciones como argumento de entrada a otra función**

**decoradores @**

**Algunos uso para decoradores**

**Decoradores con parámetros**

**Medición del Tiempo de Ejecución**

**Validación de Usuario**

**Funciones como argumento de entrada a otra función:**

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def operacion(func\_parametro, numeros):  #------------------------------------------------------------------------  # opcion 1  # ~ resultados = []  # ~ for num in numeros:  # ~ resultados.append(func\_parametro(num))  #------------------------------------------------------------------------  # opcion 2  # ~ resultados = [func\_parametro(num) for num in numeros ]  #------------------------------------------------------------------------  # opcion 3  resultados = list(map(func\_parametro,numeros) )  #------------------------------------------------------------------------  return resultados  def cuadrado(num):  return num \*\* 2  def cubo(num):  return num \*\* 3  numeros = [1, 2, 3, 4, 5]  r\_cuadrado = operacion(cuadrado, numeros)  r\_cubo = operacion(cubo, numeros)  print(f"{r\_cuadrado=}")  print(f"{r\_cubo=}") |   ·Salida esperada por consola   |  | | --- | | r\_cuadrado=[1, 4, 9, 16, 25]  r\_cubo=[1, 8, 27, 64, 125 |   · |

·

**decoradores @:**

Los decoradores son una característica poderosa y flexible que permite modificar o extender el comportamiento de funciones o métodos sin cambiar su código interno.

Los decoradores son funciones que toman otra función (o método) como argumento y devuelven una nueva función que generalmente agrega funcionalidad adicional antes o después de la función original.

Esto es especialmente útil para tareas como la verificación de permisos, el registro de actividad, la medición del tiempo de ejecución, etc.

La sintaxis básica de la función decorador() es la siguiente:

|  |
| --- |
| @<nombre de función de orden superior> |

Los decoradores se aplican utilizando la sintaxis del símbolo "@" justo antes de la definición de una función. Aquí tienes un ejemplo básico de cómo se vería un decorador y cómo se aplicaría a una función:

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def mi\_decorador(func):  def nueva\_funcion():  print("Antes de llamar a la función original")  func()  print("Después de llamar a la función original")  return nueva\_funcion  @mi\_decorador  def mi\_funcion():  print("¡Hola desde mi\_funcion!")  mi\_funcion() |   ·Salida esperada por consola   |  | | --- | | Antes de llamar a la función original  ¡Hola mundo IT!  Después de llamar a la función original |   · |

·

En este ejemplo, mi\_decorador es una función que toma otra función func como argumento y devuelve una nueva función nueva\_funcion que agrega mensajes antes y después de llamar a func. Luego, el decorador se aplica a la función mi\_funcion utilizando la sintaxis @mi\_decorador, lo que es equivalente a hacer mi\_funcion = mi\_decorador(mi\_funcion).

|  |
| --- |
| **Algunos uso para decoradores** |
| **Medición del tiempo de ejecución:**  Puedes utilizar un decorador para medir cuánto tiempo toma ejecutar una función y registrar el tiempo.  **Control de acceso y autenticación:**  Los decoradores pueden ser útiles para verificar permisos y autenticación antes de permitir que una función se ejecute.  **Caché de resultados:**  Puedes crear un decorador que almacene en caché los resultados de una función para evitar cálculos repetidos.  **Registro y seguimiento:**  Los decoradores pueden utilizarse para registrar actividad o eventos en una función, como mensajes de depuración.  **Validación de entrada:**  Un decorador puede verificar la validez de los argumentos antes de que una función se ejecute.  La versatilidad de los decoradores radica en su capacidad para modular y extender el comportamiento de las funciones sin tener que modificar su código principal. Esto ayuda a mantener un código más limpio, organizado y reutilizable. |

·

**Decoradores con parámetros:**

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def(parametro):  def mi\_decorador(func):  def nueva\_funcion():  print("Antes de llamar a la función original")  print (f"el argumento recibido {parametro}")  func()  print("Después de llamar a la función original")  return nueva\_funcion  return mi\_decorador  @decorador\_lee\_parametro("argumento")  def saludar():  print("¡Hola desde mi\_funcion!")  saludar() |   ·Salida esperada por consola   |  | | --- | | Antes de llamar a la función original  el argumento recibido argumento  ¡Hola desde mi\_funcion!  Después de llamar a la función original |   · |

·

**Medición del Tiempo de Ejecución:**

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | import time  def medir\_tiempo(func):  def wrapper(\*args, \*\*kwargs):  inicio = time.time()  resultado = func(\*args, \*\*kwargs)  fin = time.time()  print(f"Tiempo de ejecución: {fin - inicio} segundos")  return resultado  return wrapper  @medir\_tiempo  def contar\_hasta(n):  for i in range(n):  pass  print (f"valor de {n=}")  contar\_hasta(1000000) |   ·Salida esperada por consola   |  | | --- | | valor de n=1000000  Tiempo de ejecución: 0.09336233139038086 segundos |   · |

·

**Validación de Usuario:**

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def autenticar(func):  def wrapper(user, password):  if user == "admin" and password == "123":  return func(user)  else:  print(f"Acceso denegado {user}\nno puede ingresar a función")  return wrapper  @autenticar  def mostrar\_info\_segura(user):  print(f"usuario {user} aceptado.")  mostrar\_info\_segura("admin", "123") # Acceso concedido  print ("\*"\*50)  mostrar\_info\_segura("usuario", "543") |   ·Salida esperada por consola   |  | | --- | | usuario admin aceptado.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Acceso denegado usuario  no puede ingresar a función |   · |

·