Documentación curso Python

Imprimir en consola con print()

La instrucción print() es una función incorporada en Python que se utiliza para mostrar información por pantalla. Esta función recibe como argumento cualquier tipo de dato que se quiera imprimir (texto, números, resultados de cálculos, valores de variables, etc.) y lo escribe en la salida estándar, que normalmente es la consola.

```
1.  print("Hola mundo!")
```

Este código utiliza la función print() para mostrar una cadena de texto. La cadena "Hola mundo!" es un tipo de dato llamado string, y está entre comillas dobles. Python imprimirá exactamente lo que está entre comillas. Cuando se ejecuta este código, la consola mostrará: Hola mundo!.

Asignación de variables y uso de print()

La instrucción de asignación en Python utiliza el signo igual (=) para guardar un valor en una variable. Una variable es un nombre que hace referencia a un dato en memoria, y puede contener distintos tipos de valores como texto (strings) o números (enteros, flotantes, etc.). La función print() se usa para mostrar el contenido de una variable en la consola.

```
1. nombre = "Juan"
2. edad = 25
3.
4. print(nombre)
5.
6. nombre_completo = "Juan valdes"
7. nombre_primero = "Juan"
8.
```

El primer bloque asigna el texto "Juan" a la variable nombre, y el número 25 a la variable edad. Luego, con print(nombre), se muestra el contenido de la variable nombre, que en este caso es el texto Juan.

Después se crea una nueva variable llamada nombre_completo con el valor "Juan valdes", y otra llamada nombre_primero con el valor "Juan". Estas variables también almacenan cadenas de texto y permiten separar o reutilizar partes del nombre según se necesite.

Tipos de datos

En Python existen distintos tipos de datos básicos que permiten representar información de diferentes formas. Las variables se utilizan para almacenar estos datos, y la función type() permite conocer el tipo de dato almacenado. Además, es posible convertir entre tipos usando funciones como int(), float() o str().

```
1. # Diccionario -> Clave - valor | Clave tiene que ser de tipo inmutable
 2. estudiantes = {"nombre": "Carlos", "edad": 22, "carrera": "Ingeniería"}
 3.
```

```
# Enteros
variable_entera = 2024

# Flotantes
variable_flotante = 2024.33

# Cadena de texto
variable_cadena = "Hola, mundo"

# Booleanos
variable_booleano = True

# Lista
variable_lista = [1, 2, 3, 4]

# Tupla
variable_tupla = (1, 2, 3, 4)

# Diccionarios
variables_diccionario = {"nombre": "Carlos", "edad": 22, "carrera": "Ingeniería"}

print(variable_entera, variable_flotante, variable_cadena, variable_booleano,
variable_lista, variable_tupla, variables_diccionario)
print(type(variable_entera), type(variable_flotante), type(variable_cadena),
type(variable_booleano), type(variable_lista), type(variable_tupla),
type(variables_diccionario))

variable_string = "23"
variable_entero = int(variable_string)
variable_flotante = float(variable_entero)
variable_entero = int(variable_flotante)

print(variable_string, variable_flotante, variable_entero)

print(type(variable_string), type(variable_flotante), type(variable_entero))
```

Este bloque de código muestra ejemplos de los tipos de datos más comunes en Python:

- int: números enteros, como 2024.
- float: números decimales, como 2024.33.
- str: cadenas de texto, como "Hola, mundo".
- bool: valores booleanos, como True o False.
- list: listas que pueden contener varios valores, como [1, 2, 3, 4].
- tuple: tuplas, similares a listas pero inmutables, como (1, 2, 3, 4).
- dict: diccionarios que almacenan pares clave-valor, como {"nombre": "Carlos", "edad": 22}.

Las instrucciones print() muestran los valores almacenados y los tipos de cada variable. También se demuestra cómo convertir una cadena ("23") en un número entero (int) y luego a un flotante (float), lo que permite transformar y reutilizar datos según se necesite.

Cadenas de caracteres

Las cadenas de caracteres en Python (tipo str) permiten almacenar y manipular texto. Se pueden definir usando comillas simples, dobles o triples para textos multilínea. Python ofrece múltiples métodos para trabajar con cadenas, como concatenación, repetición,

conversión de mayúsculas/minúsculas, eliminación de espacios, reemplazo de palabras y separación o unión de palabras.

```
primer_nombre = "Juan"
segundo_nombre = "Alberto"

texto_largo = """
Esto es un
test para
comprobar los
textos
largos
"""

print(texto_largo)

nombre_completo = primer_nombre + " " + segundo_nombre
print(nombre_completo)

repiticion = "JaJaJa" * 5
print(repiticion)

minusculas = "hola mundo"
print(minusculas.upper())

mayusculas = "HOLA MUNDO"
print(mayusculas.lower())

datos_tabla = "Ordenadores        "
print(len(datos_tabla))
print(datos_tabla.strip())
print(len(datos_tabla.strip()))

frase = "Me gusta  Python"
print(frase)

nueva_frase = frase.replace("Python", "Java")
nueva_frase = nueva_frase.split(" ")
print(nueva_frase)

print(" ".join(nueva_frase))

```

El código comienza con la creación de dos cadenas primer_nombre y segundo_nombre, que se concatenan con un espacio entre ellas para formar el nombre_completo. También se muestra cómo definir una cadena multilínea usando triple comillas.

Se utiliza la repetición con "JaJaJa" * 5 para generar una cadena repetida. Luego, se aplican métodos como .upper() para convertir texto a mayúsculas, .lower() para convertirlo a minúsculas, y .strip() para eliminar espacios al principio y al final de la cadena.

La función len() se usa para contar caracteres. Finalmente, se realiza una sustitución de texto con .replace(), se divide una cadena en una lista de palabras usando .split(), y se reconstruye la cadena con .join().

Recibiendo datos del usuario

Python permite recibir información desde el teclado mediante la función input(). Esta función siempre devuelve el dato ingresado como una cadena de texto (tipo str). Si se necesita trabajar con números, es necesario convertir el valor utilizando funciones como int() para enteros o float() para decimales.

```
nombre = input("Como te llamas?")

edad = input("Cuantos años tienes?")

print("Hola", nombre, "tienes", edad, "años")

print(type(nombre), type(edad))

edad = int(edad)

medida = input("Cuanto mides?")

medida = float(medida)

print(type(medida))

```

Este código solicita al usuario que introduzca su nombre y su edad utilizando input(), y luego muestra un mensaje combinando texto y variables. La función type() se utiliza para verificar el tipo de dato recibido, que inicialmente es siempre str.

A continuación, la variable edad se convierte a entero con int(edad) para poder usarla como número. Lo mismo se hace con la variable medida, que se convierte a float para representar una medida decimal (por ejemplo, altura). Finalmente, se imprime el tipo de dato de medida para comprobar la conversión.

Operadores aritméticos, lógicos, relacionales y de asignación

Python incluye diferentes tipos de operadores que permiten realizar cálculos, comparaciones lógicas, verificaciones entre valores y asignaciones de variables. Estos operadores son fundamentales para construir condiciones, realizar operaciones matemáticas y controlar el flujo del programa.

```
# Operadores aritméticos

# Suma
print(2 + 2)
# Resta
print(5 - 3)
# División
print(4 / 2)
print(5 // 3)
# Multiplicación
print(4 * 2)
# Módulo
print(10 % 2)
# Potencia **
print(2 ** 2)

# Operadores lógicos True o False booleanos and or not

# And
print((5 > 3 and 2 < 4))
# OR
print((5 > 3 or 2 < 4))
# Not
print(not(5 > 3))

# Operadores relacionales o de comparación

# Igualdad ==
print(5 == 2)
```

```
# Menor o igual que <=
print(5 <= 2)
# Mayor o igual que >=
print(5 >= 2)

# Operadores de asignación
edad = 2
edad *= 2

print(edad)

```

Los **operadores aritméticos** permiten realizar cálculos matemáticos: + (suma), - (resta), / (división), // (división entera), * (multiplicación), % (módulo o resto) y ** (potencia).

Los **operadores lógicos** devuelven valores booleanos (True o False) y se usan para combinar condiciones: and (ambas verdaderas), or (al menos una verdadera) y not (niega el valor lógico).

Los **operadores relacionales** comparan dos valores: == (igual a), <= (menor o igual), >= (mayor o igual). Devuelven True o False según el resultado de la comparación.

Los **operadores de asignación** permiten modificar y guardar valores en variables. Por ejemplo, edad *= 2 multiplica el valor de edad por 2 y almacena el resultado en la misma variable.

Condicionales

Las estructuras condicionales en Python permiten ejecutar diferentes bloques de código dependiendo del valor de una condición. Se utilizan las instrucciones if, elif y else para evaluar expresiones booleanas y controlar el flujo del programa en función de diferentes escenarios.

```
nota_examen = int(input("Introduce tu nota:"))

if nota_examen < 50:
  print("Suspenso")
elif not nota_examen >= 50 and nota_examen < 70:
  print("Aprobado")
elif not nota_examen >= 70 and nota_examen < 90:
  print("Notable")
else:
  print("Excelente")

dia_actual = input("Que día es?")

if dia_actual == "Lunes":
  print("Es lunes")
elif dia_actual == "Martes":
  print("Es martes")
elif dia_actual == "Sabado" or dia_actual == "Domingo":
  print("Es fin de semana")
else:
  print("Es otro día")

```

El primer bloque de código solicita al usuario una nota mediante input() y la convierte a entero con int(). Luego evalúa el valor con una serie de condiciones if, elif y else para clasificar la nota como "Suspenso", "Aprobado", "Notable" o "Excelente". Se usan operadores relacionales (<, >=) y lógicos (not, and) para construir las condiciones.

El segundo bloque pregunta por el día de la semana y evalúa el valor introducido con condiciones if y elif utilizando el operador de igualdad (==) y el operador lógico or. Dependiendo del día, muestra un mensaje correspondiente. Si no coincide con ninguno de los días evaluados, se ejecuta la instrucción else con un mensaje genérico.

Comentarios

Los comentarios en Python se utilizan para escribir anotaciones en el código que no se ejecutan. Sirven para explicar lo que hace el programa, facilitar su comprensión o dejar notas para otros programadores (o para uno mismo en el futuro). Existen comentarios de una sola línea y de varias líneas.

```
1. # Esto es un comentario de una sola línea
2.
3. """
4. Esto son comentarios
5. de varias líneas
6. """
7.
```

La línea que comienza con # es un comentario de una sola línea. Todo lo que se escriba después del símbolo # será ignorado por el intérprete de Python.

Las comillas triples """ """ permiten escribir comentarios de varias líneas. Aunque técnicamente Python las interpreta como cadenas de texto multilínea, si no se asignan a una variable, se pueden usar como comentarios.

Listas y tuplas

Las listas y tuplas son estructuras de datos en Python que permiten almacenar colecciones de elementos. La principal diferencia es que las listas son mutables (se pueden modificar) y las tuplas son inmutables (no se pueden cambiar una vez creadas).

```
1. # 1. Crear una lista
 2. mi_lista = [1, 2, 3, 4, 5]
 3. print("Lista inicial:", mi_lista)
 4.
 5. # 2. Crear una tupla
 6. mi_tupla = (1, 2, 3, 4, 5)
 7. print("Tupla inicial:", mi_tupla)
 8.
 9. # 3. Acceder a elementos
10. print("Primer elemento de la lista:", mi_lista[0])
11. print("Primer elemento de la tupla:", mi_tupla[0])
12.
13. # 4. Modificar una lista
14. mi_lista[0] = 10
15. print("Lista después de modificar:", mi_lista)
16.
17. # 5. Intentar modificar una tupla (esto genera un error)
18. # mi_tupla[0] = 10  # Descomenta esta línea para ver el error
19.
20. # 6. Agregar elementos a una lista
21. mi_lista.append(6)
22. print("Lista después de agregar un elemento:", mi_lista)
23.
24. # 7. Extender una lista
25. mi_lista.extend([7, 8, 9])
26. print("Lista después de extender:", mi_lista)
27.
28. # 8. Convertir una tupla en una lista
```

```
tupla_a_lista = list(mi_tupla)
print("Tupla convertida en lista:", tupla_a_lista)

# 9. Convertir una lista en una tupla
lista_a_tupla = tuple(mi_lista)
print("Lista convertida en tupla:", lista_a_tupla)

# 10. Desempaquetar valores de una tupla o lista
a, b, c, *resto = mi_tupla
print("Primeros tres valores desempaquetados:", a, b, c)
print("Resto de valores:", resto)

# 11. Operaciones comunes
# Longitud
print("Longitud de la lista:", len(mi_lista))
print("Longitud de la tupla:", len(mi_tupla))

# Verificar si un elemento existe
print("¿5 está en la lista?", 5 in mi_lista)
print("¿5 está en la tupla?", 5 in mi_tupla)

# Concatenar tuplas o listas
nueva_tupla = mi_tupla + (6, 7)
print("Tupla concatenada:", nueva_tupla)
nueva_lista = mi_lista + [10, 11]
print("Lista concatenada:", nueva_lista)

# 12. Repetir elementos
print("Tupla repetida:", mi_tupla * 2)
print("Lista repetida:", mi_lista * 2)

```

El código comienza creando una lista y una tupla. Ambas estructuras permiten acceder a sus elementos por índice. Las listas se pueden modificar (por ejemplo, cambiando un valor o agregando nuevos elementos con append() o extend()), mientras que las tuplas no permiten cambios directos, lo que genera un error si se intenta.

También se muestra cómo convertir entre listas y tuplas con las funciones list() y tuple(). La técnica de **desempaquetado** permite asignar los primeros valores a variables individuales y el resto a una lista usando el operador *.

Además, se presentan operaciones comunes como obtener la longitud con len(), verificar si un elemento está contenido con in, concatenar con +, y repetir los elementos con el operador *.

Diccionarios

Los diccionarios en Python permiten almacenar datos en forma de pares clave-valor. Las claves deben ser de tipo inmutable (como cadenas o números), y se utilizan para acceder a sus valores asociados. Los diccionarios son mutables, lo que significa que se pueden modificar después de ser creados.

```
# Crear un diccionario básico
mi_diccionario = {
    "nombre": "Juan",
    "edad": 30,
    "ocupacion": "Ingeniero"
}

# Acceder a valores usando claves
print(mi_diccionario["nombre"])  # Salida: Juan

# Agregar un nuevo par clave-valor
mi_diccionario["ciudad"] = "Madrid"
```

```
print(mi_diccionario)

# Modificar un valor existente
mi_diccionario["edad"] = 31
print(mi_diccionario)

# Eliminar un par clave-valor
del mi_diccionario["ocupacion"]
print(mi_diccionario)

# Verificar si una clave existe
if "nombre" in mi_diccionario:
    print("La clave 'nombre' existe.")

# Iterar sobre un diccionario
for clave, valor in mi_diccionario.items():
    print(f"{clave}: {valor}")

# Obtener solo las claves
print(mi_diccionario.keys())

# Obtener solo los valores
print(mi_diccionario.values())

# Obtener solo los pares clave-valor
print(mi_diccionario.items())

# Copiar un diccionario
copia_diccionario = mi_diccionario.copy()
print(copia_diccionario)

# Limpiar un diccionario
mi_diccionario.clear()
print(mi_diccionario)  # Salida: {}

```

El código comienza creando un diccionario con tres pares clave-valor. Se accede al valor asociado a una clave usando corchetes. Se agregan nuevos elementos asignando una nueva clave, y se modifican valores existentes asignando un nuevo valor a una clave ya definida.

Se puede eliminar una entrada con del, verificar si una clave existe con el operador in, e iterar sobre el diccionario con un bucle for usando .items() para obtener tanto claves como valores.

También se puede acceder solo a las claves con .keys(), a los valores con .values(), o a ambos con .items(). El método .copy() permite duplicar un diccionario, y .clear() lo vacía completamente.

Ciclos

Los ciclos permiten ejecutar repetidamente un bloque de código mientras se cumpla una condición o durante un número determinado de veces. Python ofrece dos tipos principales de ciclos: for (para iterar sobre secuencias o rangos) y while (mientras se cumpla una condición). También se pueden usar instrucciones como break, continue y else para controlar el flujo de ejecución dentro de los ciclos.

```
# Ejemplo 1: Ciclo for
# Itera sobre una lista
frutas = ["manzana", "plátano", "cereza"]
for fruta in frutas:
    print(fruta)

```

```python
# Ejemplo 2: Ciclo for con range
# Itera un rango de números
for numero in range(5):  # Imprime del 0 al 4
    print(numero)

# Ejemplo 3: Ciclo while
# Ejecuta mientras la condición sea verdadera
contador = 0
while contador < 5:
    print(contador)
    contador += 1  # Incrementa el contador

# Ejemplo 4: Uso de break
# Termina el ciclo si se cumple una condición
for letra in "Python":
    if letra == "h":
        break  # Sale del ciclo al encontrar "h"
    print(letra)

# Ejemplo 5: Uso de continue
# Salta a la siguiente iteración
for numero in range(6):
    if numero % 2 == 0:  # Salta los números pares
        continue
    print(numero)

# Ejemplo 6: Ciclos anidados
# Iteración de dos ciclos
for i in range(3):  # Ciclo externo
    for j in range(3):  # Ciclo interno
        print(f"i={i}, j={j}")

# Ejemplo 7: else con ciclos
# Ejecuta el bloque else si el ciclo termina normalmente
for numero in range(3):
    print(numero)
else:
    print("Ciclo completado")

# Ejemplo 8: Ciclo infinito (¡Ten cuidado!)
# while True:
#     print("Esto se repetirá para siempre")
#     break  # Asegúrate de tener una condición de salida

```

El primer bloque muestra un ciclo for que recorre una lista de frutas. Luego, se utiliza range() para iterar sobre una secuencia de números del 0 al 4. En el ejemplo con while, el ciclo se repite mientras la condición contador < 5 sea verdadera, incrementando el valor en cada iteración.

El uso de break permite salir de un ciclo anticipadamente cuando se cumple una condición. Por otro lado, continue omite la iteración actual y pasa a la siguiente. En los ciclos anidados se combinan dos bucles for para generar pares de valores.

La cláusula else puede acompañar a un ciclo para ejecutar un bloque de código solo si el ciclo no fue interrumpido por un break. Finalmente, se muestra cómo se puede crear un ciclo infinito con while True, que solo se detiene si se incluye una condición de salida, como un break.

Funciones

Las funciones en Python permiten agrupar bloques de código reutilizables. Se definen con la palabra clave def, pueden recibir parámetros de entrada y devolver valores con return.

También existen funciones anónimas con lambda, funciones con parámetros opcionales y aquellas que aceptan un número variable de argumentos.

```
1. # Ejemplo 1: Definir y llamar una función simple
 2. def saludar():
 3.     print("¡Hola, mundo!")
 4.
 5. saludar()
 6.
 7. # Ejemplo 2: Función con parámetros
 8. def saludar_persona(nombre):
 9.     print(f"¡Hola, {nombre}!")
10.
11. saludar_persona("Ana")
12.
13. # Ejemplo 3: Función con retorno de valor
14. def sumar(a, b):
15.     return a + b
16.
17. resultado = sumar(5, 7)
18. print(f"La suma es: {resultado}")
19.
20. # Ejemplo 4: Función con parámetros predeterminados
21. def presentar(nombre, edad=18):
22.     print(f"Nombre: {nombre}, Edad: {edad}")
23.
24. presentar("Carlos")
25. presentar("Laura", 25)
26.
27. # Ejemplo 5: Función con argumentos variables
28. def listar_amigos(*amigos):
29.     print("Mis amigos son:")
30.     for amigo in amigos:
31.         print(f"- {amigo}")
32.
33. listar_amigos("Juan", "María", "Pedro")
34.
35. # Ejemplo 6: Función con argumentos clave-valor variables
36. def describir_persona(**info):
37.     for clave, valor in info.items():
38.         print(f"{clave}: {valor}")
39.
40. describir_persona(nombre="Lucía", edad=30, ocupación="Ingeniera")
41.
42. # Ejemplo 7: Función lambda (función anónima)
43. doblar = lambda x: x * 2
44. print(doblar(10))
45.
46. # Ejemplo 8: Función como parámetro
47. def operar(a, b, operacion):
48.     return operacion(a, b)
49.
50. resultado = operar(10, 5, lambda x, y: x - y)
51. print(f"El resultado de la operación es: {resultado}")
52.
```

El primer ejemplo define una función sin parámetros que imprime un saludo al ser llamada. Luego, se muestra cómo definir una función con un parámetro (nombre) para personalizar el saludo. En el tercer ejemplo, la función sumar devuelve el resultado de una operación utilizando return.

También se muestran funciones con parámetros predeterminados, que asignan un valor por defecto si no se proporciona uno. Las funciones con *args aceptan una cantidad variable de argumentos como una tupla, mientras que **kwargs permite recibir múltiples parámetros con nombre en forma de diccionario.

Las funciones lambda son funciones anónimas útiles para operaciones simples. Finalmente, se muestra cómo pasar funciones como parámetros a otras funciones, permitiendo operaciones dinámicas.

Recursividad

La recursividad es una técnica de programación en la que una función se llama a sí misma para resolver un problema dividiéndolo en subproblemas más pequeños. Toda función recursiva debe tener al menos un **caso base**, que es la condición que detiene las llamadas recursivas, evitando así bucles infinitos.

```
1. def factorial(n):
 2.     # Caso base: Si n es 0 o 1, el factorial es 1
 3.     if n == 0 or n == 1:
 4.         return 1
 5.     # Caso recursivo: n * factorial(n-1)
 6.     else:
 7.         return n * factorial(n - 1)
 8.
 9. # Probar la función
10. print(factorial(5))  # Salida: 120
11.
12. # Ejemplo simple de recursividad: Serie de Fibonacci
13. def fibonacci(n):
14.     # Caso base: Fibonacci(0) = 0, Fibonacci(1) = 1
15.     if n == 0:
16.         return 0
17.     elif n == 1:
18.         return 1
19.     # Caso recursivo: Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
20.     else:
21.         return fibonacci(n - 1) + fibonacci(n - 2)
22.
23. # Probar la función
24. for i in range(10):
25.     print(fibonacci(i), end=" ")  # Salida: 0 1 1 2 3 5 8 13 21 34
26.
```

La función factorial(n) calcula el factorial de un número de forma recursiva. Si n es 0 o 1, devuelve 1 (caso base). Si no, multiplica n por el resultado de llamar a sí misma con n - 1 (caso recursivo).

La función fibonacci(n) devuelve el n-ésimo término de la serie de Fibonacci. Tiene dos casos base (n == 0 y n == 1) y un caso recursivo que suma los dos valores anteriores (fibonacci(n-1) + fibonacci(n-2)). Un ciclo for imprime los primeros 10 términos de la serie.

Archivos

Python permite trabajar con archivos para leer y escribir datos. La función open() se utiliza para abrir archivos, y con el modo "w" se abre el archivo en modo escritura, creando uno nuevo si no existe o sobrescribiéndolo si ya existe. El bloque with se usa para abrir archivos de forma segura, cerrándolos automáticamente al finalizar.

```
1. # Definir el array con las líneas de texto
2. lineas = ["Línea 1: Hola, mundo.", "Línea 2: Esto es Python.", "Línea 3: ¡Es fácil
crear archivos!"]
```

```
3.
4. # Abrir un archivo en modo escritura ('w') y escribir las líneas
5. with open("archivo.txt", "w") as archivo:
6.     for linea in lineas:
7.         archivo.write(linea + "\n")
8.
```

El código define una lista de cadenas de texto llamada lineas. Luego, se abre un archivo llamado archivo.txt en modo escritura usando with open(...). Dentro del bloque, se recorre cada elemento de la lista con un ciclo for y se escribe en el archivo, agregando un salto de línea \n al final de cada línea. Al finalizar, el archivo se guarda automáticamente y se cierra sin necesidad de usar archivo.close().

Importaciones

En Python, las importaciones permiten utilizar módulos y librerías externas que contienen funciones, clases y herramientas adicionales. La instrucción import se usa para acceder a estos recursos y facilitar tareas específicas como cálculos numéricos, visualización de datos o procesamiento de imágenes.

python

CopyEdit

import numpy as np

import matplotlib.pyplot as plt

import cv2

- import numpy as np: importa la librería **NumPy**, especializada en operaciones con arrays y cálculos matemáticos. Se usa el alias np para acceder a sus funciones de forma más corta.
- import matplotlib.pyplot as plt: importa el módulo pyplot de la librería **matplotlib**, utilizado para crear gráficos y visualizaciones. Se accede mediante el alias plt.
- import cv2: importa la librería **OpenCV**, ampliamente utilizada en visión por computadora para trabajar con imágenes y vídeos.

Estas importaciones permiten ampliar las capacidades del lenguaje Python mediante herramientas potentes y especializadas.

Errores y excepciones

En Python, los errores que ocurren durante la ejecución del programa se denominan **excepciones**. Para evitar que el programa se detenga ante un error, se pueden capturar y gestionar usando bloques try, except, else y finally. También es posible definir excepciones personalizadas.

```
1. # Ejemplo básico de manejo de excepciones
 2. try:
 3.     numero = int(input("Ingresa un número: "))
```

```
    print(f"El número ingresado es {numero}")
except ValueError:
    print("Error: Debes ingresar un número válido.")

# Manejo de múltiples excepciones
try:
    dividendo = int(input("Ingresa el dividendo: "))
    divisor = int(input("Ingresa el divisor: "))
    resultado = dividendo / divisor
    print(f"El resultado es {resultado}")
except ValueError:
    print("Error: Debes ingresar números válidos.")
except ZeroDivisionError:
    print("Error: No se puede dividir entre cero.")

# Uso de 'else' y 'finally'
try:
    archivo = open("mi_archivo.txt", "r")
    contenido = archivo.read()
    print("Archivo leído correctamente.")
except FileNotFoundError:
    print("Error: El archivo no existe.")
else:
    print("El archivo se procesó correctamente.")
finally:
    print("Cerrando el archivo.")
    archivo.close()

# Creando excepciones personalizadas
class MiErrorPersonalizado(Exception):
    def __init__(self, mensaje):
        super().__init__(mensaje)

try:
    valor = int(input("Ingresa un número mayor que 10: "))
    if valor <= 10:
        raise MiErrorPersonalizado("El valor debe ser mayor que 10.")
except MiErrorPersonalizado as e:
    print(f"Error personalizado: {e}")

```

En el primer bloque se utiliza try para intentar convertir una entrada a entero. Si el usuario introduce un valor no numérico, se lanza una excepción ValueError que se captura con except.

En el segundo bloque se gestionan múltiples tipos de errores: ValueError si la entrada no es un número y ZeroDivisionError si se intenta dividir entre cero.

El tercer bloque muestra cómo usar else para ejecutar código si no hubo error, y finally para ejecutar código sin importar si hubo o no una excepción (por ejemplo, cerrar un archivo).

Por último, se define una **excepción personalizada** creando una clase que hereda de Exception. Si el valor introducido no cumple con la condición, se lanza un error con raise, y se captura con except mostrando un mensaje específico.

Programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación que organiza el código en torno a **clases** y **objetos**. Las clases definen la estructura y comportamiento de los objetos, y los objetos son instancias concretas de esas clases.

```python
# Definición de la clase Perro
class Perro:
    # Método constructor para inicializar los atributos de la clase
    def __init__(self, nombre, raza):
        self.nombre = nombre
        self.raza = raza

    # Método para que el perro salude
    def saludar(self):
        return f"¡Guau! Soy {self.nombre}, un {self.raza}."

# Creación de una instancia de la clase Perro
mi_perro = Perro("Max", "Golden Retriever")

# Llamada al método saludar
print(mi_perro.saludar())

```

En este ejemplo se define una clase llamada Perro con dos elementos principales:

- Un **método constructor** __init__() que se ejecuta al crear una nueva instancia de la clase. Recibe los parámetros nombre y raza y los asigna como atributos del objeto.
- Un **método** llamado saludar() que devuelve un mensaje personalizado con los atributos del perro.

Luego, se crea un objeto mi_perro utilizando la clase Perro, y se llama al método saludar() para obtener un mensaje. La palabra clave self se refiere a la instancia actual del objeto y permite acceder a sus atributos y métodos.

Interfaces gráficas

Python permite crear interfaces gráficas de usuario (GUI) usando la librería **tkinter**. Esta librería proporciona herramientas para construir ventanas, botones, cuadros de texto y otros elementos interactivos. La ventana principal se gestiona mediante un bucle (mainloop()) que mantiene la aplicación en ejecución.

```python
import tkinter as tk
from tkinter import ttk  # Para estilos adicionales

# Crear la ventana principal
root = tk.Tk()
root.title("Ejemplo de Widgets Tkinter")
root.geometry("600x600")

# --- 1. Label ---
label = tk.Label(root, text="Widget de label",
                 font=("Helvetica", 14, "bold"),  # Fuente
                 fg="white", bg="blue",  # Colores de texto y fondo
                 padx=10, pady=10,  # Padding (relleno interno)
                 borderwidth=5, relief="solid"  # Borde
                 )
label.pack(pady=20)  # Empaque en la ventana principal con padding

# --- 2. Button ---
def button_click():
    print("¡Botón presionado!")

button = tk.Button(root, text="Presiona aquí",
                   font=("Arial", 12),  # Fuente
```

```
                    fg="white", bg="green",  # Colores de texto y fondo
                    activebackground="darkgreen",  # Color cuando está presionado
                    activeforeground="yellow",  # Color del texto cuando está
presionado
                    padx=10, pady=5,  # Padding del botón
                    borderwidth=3, relief="raised",  # Estilo del borde
                    command=button_click)  # Acción al presionar
button.pack(pady=20)

# --- 3. Entrada de texto ---
entry = tk.Entry(root, font=("Arial", 12),
                 fg="black", bg="lightgray",  # Colores de texto y fondo
                 justify="center",  # Alineación del texto dentro del Entry
                 width=20,  # Ancho del Entry en caracteres
                 borderwidth=2, relief="groove"  # Estilo del borde
                 )
entry.pack(pady=20)

# --- 4. Marcos ---
marco = tk.Frame(root, bg="lightblue",
                 borderwidth=3, relief="ridge",  # Borde del marco
                 padx=10, pady=10  # Padding interno del marco
                 )
marco.pack(pady=20, fill="both", expand=True)  # Empaque del marco con expansión

# Agregar widgets dentro del Marco
label_in_marco = tk.Label(marco, text="Label dentro de un marco", bg="lightblue",
font=("Arial", 10))
label_in_marco.pack(pady=5)

button_in_marco = tk.Button(marco, text="Botón dentro de un marco", command=lambda:
print("Botón dentro de un marco"))
button_in_marco.pack(pady=5)

# Iniciar el loop de la ventana principal
root.mainloop()

```

Este código construye una ventana con diferentes elementos gráficos:

- Label: muestra texto en la ventana con estilo personalizado.
- Button: crea un botón que ejecuta una acción al hacer clic, definida por la función command.
- Entry: permite ingresar texto desde el usuario.
- Frame: crea un contenedor que agrupa otros widgets, útil para organizar visualmente la interfaz.

Cada widget se coloca en la ventana con .pack() y se pueden configurar estilos visuales como fuentes, colores, bordes y alineaciones. La ejecución finaliza con root.mainloop(), que mantiene abierta la interfaz y gestiona eventos como clics y entradas de texto.

Flask

Flask es un framework de desarrollo web **minimalista y flexible** para Python. Se describe como un *microframework* porque no incluye todas las herramientas o funcionalidades integradas que tienen otros frameworks más grandes como Django. Sin embargo, es extremadamente poderoso y adaptable, lo que lo hace ideal tanto para **aplicaciones pequeñas** como para **proyectos más grandes**.

Además de facilitar la creación de aplicaciones web, Flask también se utiliza para **lanzar servidores web** de manera rápida y sencilla. Esto permite que los desarrolladores prueben sus aplicaciones en un entorno local antes de desplegarlas en producción.

Características principales de Flask

Ligero y minimalista

- No impone una estructura fija ni un patrón de diseño específico.
- Permite construir aplicaciones desde cero, eligiendo solo las herramientas necesarias.
- Ideal para proyectos que no requieren un framework grande.

Extensible

- Puede ampliarse con extensiones para agregar funcionalidades como:
    - Manejo de bases de datos (ej. *SQLAlchemy*).
    - Autenticación de usuarios.
    - Gestión de formularios.

Servidor de desarrollo integrado

- Incluye un servidor web que permite probar la aplicación localmente durante el desarrollo.

Sistema de plantillas Jinja2

- Utiliza *Jinja2*, un motor de plantillas potente que permite generar HTML dinámico y reutilizable.

Compatibilidad con WSGI

- Basado en *WSGI* (*Web Server Gateway Interface*), el estándar para aplicaciones web en Python.

Manejo de rutas

- Proporciona una manera sencilla y directa de definir rutas (URLs) dentro de la aplicación.

¿Para qué sirve Flask?

Flask sirve para construir **aplicaciones web** de todo tipo, desde pequeños prototipos hasta sistemas empresariales complejos. Algunos ejemplos de uso son:

Aplicaciones web simples

- Perfecto para mostrar páginas estáticas o contenido básico.

APIs RESTful

- Muy usado para crear APIs ligeras y rápidas que puedan ser consumidas por otras aplicaciones.

Prototipos rápidos

- Ideal para construir versiones preliminares de aplicaciones de forma ágil.

Aplicaciones dinámicas

- Permite crear aplicaciones que interactúan con bases de datos, gestionan usuarios y manejan lógica compleja.

Aplicaciones de ciencia de datos

- Usado frecuentemente para mostrar resultados de análisis de datos, gráficos o modelos de *Machine Learning* mediante interfaces web.

```
1. # Importar la librería de Flask
2. from flask import Flask, request
3. 
4. # Crear una instancia de nuestra aplicación de Flask
5. app = Flask(__name__)
6. 
7. # Ruta de la página principal
8. @app.route('/')
9. def inicio():
10.     # Página principal con un mensaje de bienvenida
11.     return '''
12.         <h1> Bienvenido a la app de flask </h1>
13.         <p> Explora las rutas de la app:</p>
14.         <ul>
15.             <li><a href="/acerca">Acerca de la app </a></li>
16.             <li><a href="/servicios">servicios de la app </a></li>
17.             <li><a href="/contacto">Contacto de la app </a></li>
18.         </ul>
19.     '''
20. 
21. @app.route("/acerca")
22. def acerca():
23.     return '''
24.         <h1>Acerca de la app de flask</h1>
25.         <p>Este es un párrafo de ejemplo</p>
26.         <p><a href="/"> Regresar al inicio</a></p>
27.     '''
28. 
29. @app.route("/servicios")
30. def servicios():
31.     return '''
32.         <h1>Servicios de la app de flask</h1>
33.         <p>Este es un párrafo de ejemplo</p>
34.         <p><a href="/"> Regresar al inicio</a></p>
35.     '''
36. 
37. @app.route("/contacto", methods=['GET', 'POST'])
38. def contacto():
39.     if request.method == 'POST':
40.         # Capturar los datos enviados por el usuario
41.         nombre = request.form['nombre']
42.         mensaje = request.form['mensaje']
43. 
44.         # Responder agradeciendo el contacto
45.         return f'''
46.             <h1>Gracias, {nombre}</h1>
47.             <p>Hemos recibido tu mensaje:</p>
48.             <blockquote>{mensaje}</blockquote>
49.             <p><a href="/">Regresar al Inicio</a></p>
```

```
        '''
    return '''
        <h1>Contáctanos</h1>
        <form method="post">
            <label> Nombre: </label><br>
            <input type="text" name="nombre" required><br><br>
            <label> Mensaje: </label><br>
            <textarea name="mensaje" required></textarea><br><br>
            <input type="submit" value="Enviar">
        </form>
        <p><a href="/"> Regresar al inicio</a></p>
    '''

@app.errorhandler(404)
def pagina_no_encontrada(error):
    return '''
    <h1>Error 404</h1>
    <p>Lo sentimos, la página que buscas no existe.</p>
    <p><a href="/">Regresar al Inicio</a></p>
''', 404

if __name__ == "__name__":
    app.run(debug=True)

```

Este código define una aplicación web básica con Flask. Se crean varias rutas (/, /acerca, /servicios, /contacto) que devuelven contenido HTML. La ruta /contacto permite enviar un formulario mediante el método POST y mostrar una respuesta personalizada.

Se maneja un error 404 con @app.errorhandler(404) para mostrar un mensaje amigable si el usuario accede a una página no existente. Finalmente, la línea app.run(debug=True) inicia el servidor en modo desarrollo, aunque contiene un error: la condición correcta es if __name__ == "__main__":.

Bases de Datos y Bases de Datos SQL

1. ¿Qué es una base de datos?

Una base de datos es una colección organizada de datos que permite almacenar, recuperar y gestionar información de manera eficiente. Los datos se estructuran para facilitar su acceso, manipulación y actualización.

Ejemplo:
Un sistema de gestión de biblioteca podría tener una base de datos que almacena información sobre libros, usuarios y préstamos.

---

2. Tipos de bases de datos

Existen varios tipos de bases de datos, pero dos categorías principales son:

- Bases de datos relacionales (SQL):
  Los datos se organizan en tablas (similares a hojas de cálculo) que están relacionadas entre sí mediante claves primarias y foráneas. Usan el lenguaje SQL (Structured Query Language) para realizar operaciones.

- Bases de datos no relacionales (NoSQL):
  Diseñadas para manejar datos no estructurados o semi-estructurados, como JSON, documentos, grafos, etc.

En esta explicación, nos centraremos en las bases de datos relacionales.

---

3. ¿Qué es SQL?

SQL (Structured Query Language) es un lenguaje estándar diseñado para interactuar con bases de datos relacionales. Con SQL puedes:

- Crear tablas y esquemas: Definir cómo se organizarán los datos.
- Insertar datos: Agregar nuevos registros a las tablas.
- Consultar datos: Recuperar información específica usando sentencias SELECT.
- Actualizar datos: Modificar registros existentes con UPDATE.
- Eliminar datos: Borrar registros con DELETE.

SQL es poderoso porque permite manejar grandes cantidades de datos de forma estructurada y eficiente.

```python
# Importar la librería
import sqlite3

# Definir una función para establecer la conexión con la base de datos
def crear_conexion(nombre_db):
  return sqlite3.connect(nombre_db)

# Definir una función para crear una tabla dentro de nuestra base de datos
def crear_tabla(conexion, nombre_tabla):
  consulta = f"""
  CREATE TABLE IF NOT EXISTS {nombre_tabla} (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nombre TEXT NOT NULL,
    edad INTEGER
  );
  """
  cursor = conexion.cursor()
  cursor.execute(consulta)
  conexion.commit()

# Definir una función para insertar datos dentro de la tabla creada
def insertar_entradas(conexion, nombre_tabla, nombre, edad):
  consulta = f"INSERT INTO {nombre_tabla} (nombre, edad) VALUES (?,?)"
  cursor = conexion.cursor()
  cursor.execute(consulta, (nombre, edad))
  conexion.commit()

# Definir una función para recuperar todas las entradas de una tabla
def recuperar_entradas(conexion, nombre_tabla):
  consulta = f"SELECT * FROM {nombre_tabla}"
  cursor = conexion.cursor()
  cursor.execute(consulta)
  return cursor.fetchall()

# Definir una función para actualizar la información de un usuario dentro de la tabla
def actualizar_entrada(conexion, nombre_tabla, id, nombre, edad):
  consulta = f"UPDATE {nombre_tabla} SET nombre = ?, edad= ? WHERE id = ?"
  cursor = conexion.cursor()
  cursor.execute(consulta, (nombre, edad, id))
```

```
    conexion.commit()

# Definir una función para eliminar un usuario dentro de nuestra tabla
def eliminar_entrada(conexion, nombre_tabla, id):
    consulta = f"DELETE FROM {nombre_tabla} WHERE id = ?"
    cursor = conexion.cursor()
    cursor.execute(consulta, (id,))
    conexion.commit()

nombre_tabla = "usuarios"
# Crear la conexión y la base de datos
conexion = crear_conexion(":memory:")
# Crear una tabla dentro de la base de datos
crear_tabla(conexion, nombre_tabla)

# Añadir entradas en la tabla
insertar_entradas(conexion, nombre_tabla, "Pedro", 30)
insertar_entradas(conexion, nombre_tabla, "Juan", 25)
insertar_entradas(conexion, nombre_tabla, "Manuel", 20)

# Recuperamos las entradas de nuestra tabla
print("Usuarios dentro de la tabla de usuarios")
usuarios = recuperar_entradas(conexion, nombre_tabla)
for usuario in usuarios:
    print(usuario)

# Modificar usuario con ID == 2
actualizar_entrada(conexion, nombre_tabla, 2, "Alberto", 23)

# Recuperamos las entradas de nuestra tabla una vez actualizadas
print("Usuarios dentro de la tabla de usuarios una vez actualizados")
usuarios = recuperar_entradas(conexion, nombre_tabla)
for usuario in usuarios:
    print(usuario)

# Eliminar un usuario
eliminar_entrada(conexion, nombre_tabla, 1)

# Recuperamos las entradas de nuestra tabla una vez eliminadas
print("Usuarios dentro de la tabla de usuarios una vez actualizados")
usuarios = recuperar_entradas(conexion, nombre_tabla)
for usuario in usuarios:
    print(usuario)

# Cerrar la conexión al finalizar
conexion.close()

```

Este programa usa SQLite para crear una base de datos en memoria (:memory:), útil para pruebas. Se define una tabla llamada usuarios con campos id, nombre y edad.

Las funciones permiten:

- Conectar con la base de datos.
- Crear la tabla si no existe.
- Insertar registros.
- Recuperar todos los registros (SELECT *).
- Actualizar datos usando UPDATE.
- Eliminar registros con DELETE.

Después de cada operación, los resultados se imprimen para verificar el estado actual de la tabla. Finalmente, se cierra la conexión con conexion.close().

Bases de datos con MongoDB y PyMongo

MongoDB es una base de datos NoSQL orientada a documentos. En Python, se puede usar la librería **PyMongo** para conectarse, insertar, consultar, actualizar y eliminar datos en colecciones. Los documentos se representan como diccionarios de Python.

```
1. # Importar las librerías
 2. from pymongo import MongoClient
 3. from pymongo.server_api import ServerApi
 4.
 5. # Crear la URL de conexión a MongoDB Atlas
 6. URL_MONGO =
"mongodb+srv://a1999mbs:1NclhX7yvLJAiCwf@ejemplo.suetq.mongodb.net/?retryWrites=true&w=maj
ority&appName=ejemplo"
 7.
 8. # Crear la conexión con MongoDB Atlas
 9. cliente = MongoClient(URL_MONGO, server_api=ServerApi('1'))
10.
11. # Crear una base de datos
12. base_datos = cliente["ejemplo"]
13.
14. # Crear una colección
15. coleccion = base_datos["usuarios"]
16.
17. # (Opcional) limpiar información de posibles conexiones anteriores
18. coleccion.delete_many({})
19.
20. # Definir una función para insertar un usuario
21. def insertar_usuario(nombre, edad, correo):
22.   documento = {"nombre": nombre, "edad": edad, "correo": correo}
23.   resultado = coleccion.insert_one(documento)
24.   return resultado.inserted_id
25.
26. # Definir una función para obtener todos los usuarios
27. def obtener_usuarios():
28.   return list(coleccion.find({}, {"_id": 0}))
29.
30. # Función para buscar un usuario por nombre
31. def buscar_usuario_por_nombre(nombre):
32.   return coleccion.find_one({"nombre": nombre}, {"_id": 0})
33.
34. # Función para actualizar la edad de un usuario
35. def actualizar_edad(nombre, nueva_edad):
36.   filtro = {"nombre": nombre}
37.   actualizacion = {"$set": {"edad": nueva_edad}}
38.   resultado = coleccion.update_one(filtro, actualizacion)
39.   return resultado.modified_count
40.
41. # Función para borrar un usuario por correo
42. def borrar_usuario_por_correo(correo):
43.   filtro = {"correo": correo}
44.   resultado = coleccion.delete_one(filtro)
45.   return resultado.deleted_count
46.
47. # Hacer una demostración de uso de la base de datos
48. # Insertar usuarios
49. print("Insertando usuarios...")
50. id1 = insertar_usuario("Juan", 23, "juan@gmail.com")
51. id2 = insertar_usuario("Alberto", 22, "alberto@gmail.com")
52. id3 = insertar_usuario("Luis", 20, "luis@gmail.com")
53. print(f"Usuarios insertados en la base de datos con los id: {id1}, {id2}, {id3}")
54.
55. # Mostrar todos los usuarios
56. print("Mostrando todos los usuarios de la base de datos")
```

```
usuarios = obtener_usuarios()
for usuario in usuarios:
  print(usuario)

# Buscar un usuario por nombre
print("Buscando usuario por nombre")
nombre_buscar = "Juan"
usuario_encontrado = buscar_usuario_por_nombre(nombre_buscar)
print(usuario_encontrado if usuario_encontrado else "Usuario no encontrado")

# Actualizar edad de un usuario
nueva_edad = 35
print("Actualizando la edad de un usuario")
usuario_modificado = actualizar_edad(nombre_buscar, nueva_edad)
print(f"Entradas modificadas: {usuario_modificado}")

# Verificar modificación
print("Buscando usuario por nombre después de modificar")
usuario_encontrado = buscar_usuario_por_nombre(nombre_buscar)
print(usuario_encontrado if usuario_encontrado else "Usuario no encontrado")

# Eliminar un usuario
correo_eliminar = "luis@gmail.com"
print("Eliminando usuario por correo")
eliminado = borrar_usuario_por_correo(correo_eliminar)
print(f"Usuarios eliminados: {eliminado}")

# Mostrar todos los usuarios después de la eliminación
print("Mostrando todos los usuarios de la base de datos después de eliminar a uno")
usuarios = obtener_usuarios()
for usuario in usuarios:
  print(usuario)

```

Este programa se conecta a una base de datos en **MongoDB Atlas** utilizando una URL de conexión. Luego, se crean funciones para gestionar una colección llamada usuarios. Estas funciones permiten:

- Insertar documentos (insert_one).
- Consultar todos los usuarios (find).
- Buscar por nombre (find_one).
- Actualizar datos con $set y update_one.
- Eliminar usuarios (delete_one).

El programa imprime el estado de la base de datos en cada etapa para verificar que las operaciones se ejecutan correctamente.

Requests en Python

Cuando trabajamos con APIs o servicios web en Python, el objetivo principal es enviar y recibir datos desde un servidor utilizando el protocolo HTTP. La biblioteca más popular y sencilla para manejar esto es requests. A continuación, se explican los conceptos básicos necesarios para entender cómo funcionan las solicitudes HTTP y cómo implementarlas en Python.

¿Qué es HTTP y cómo funcionan las requests?

- HTTP (HyperText Transfer Protocol) es el protocolo utilizado para la comunicación entre clientes (como tu navegador o una aplicación) y servidores web.
- El cliente envía una request (solicitud) al servidor, y el servidor responde con una response (respuesta).

Cada solicitud HTTP tiene:

1. Método HTTP: Define el tipo de operación que deseas realizar.
2. URL: Es la dirección del recurso al que deseas acceder.
3. Encabezados (Headers): Información adicional que se proporciona al servidor (como autenticación o tipo de datos).
4. Cuerpo (Body): Datos que se envían en algunas solicitudes, como en POST o PUT.

Métodos HTTP principales

GET

- Usado para obtener datos de un servidor.
- No modifica nada en el servidor.
- Ejemplo: Obtener los datos de un usuario.

POST

- Usado para enviar datos al servidor.
- Comúnmente utilizado para crear nuevos recursos.
- Ejemplo: Crear una nueva publicación en un blog.

PUT

- Usado para actualizar datos existentes en el servidor.
- Ejemplo: Actualizar la información de un usuario.

DELETE

- Usado para eliminar un recurso en el servidor.
- Ejemplo: Borrar un comentario.

¿Cómo funcionan las requests en Python?

La biblioteca requests simplifica el trabajo con solicitudes HTTP. Puedes realizar cualquiera de los métodos HTTP con funciones como requests.get(), requests.post(), etc. Además, se encarga automáticamente de manejar:

- Codificación de datos.
- Procesamiento de respuestas.

- Manejo de errores comunes.

Estructura de una solicitud HTTP en requests

URL

- Especifica la dirección del servidor y el recurso.
- Ejemplo: https://jsonplaceholder.typicode.com/posts.

Headers (Encabezados)

- Información adicional que el cliente envía al servidor.
- Ejemplo: Tipo de contenido, autenticación.

Cuerpo (Body)

- Usado en métodos como POST o PUT para enviar datos al servidor.
- Normalmente se envían en formato JSON.

Parámetros de consulta (Query Parameters)

- Datos adicionales enviados en la URL para filtrar o modificar la respuesta.
- Ejemplo: ?userId=1 en la URL.

Tipos de respuestas del servidor

Código de estado HTTP

Indica si la solicitud fue exitosa o no. Algunos ejemplos comunes:

- 200 OK: Solicitud exitosa.
- 201 Created: Recurso creado.
- 400 Bad Request: Error en la solicitud.
- 401 Unauthorized: Falta de autenticación.
- 404 Not Found: Recurso no encontrado.
- 500 Internal Server Error: Error en el servidor.

Cuerpo de la respuesta

- Contiene los datos que el servidor devuelve.
- Normalmente está en formato JSON.

Encabezados de la respuesta

- Información adicional del servidor, como tipo de contenido, tiempo de respuesta, etc.

```python
# Ejemplo 1: Haciendo una solicitud GET básica

# Definir la URL
url = "https://jsonplaceholder.typicode.com/posts/1"

# Hacer la petición GET
respuesta = requests.get(url)

# Verificar la respuesta del servidor
if respuesta.status_code == 200:
  print("Respuesta recibida correctamente!")
  print(respuesta.json())
else:
  print(f"Fallo en la petición GET: {respuesta.status_code}")

# Ejemplo 2: Añadiendo parámetros a nuestra consulta GET

# Definir URL
url = "https://jsonplaceholder.typicode.com/posts"

# Escribir los parámetros a enviar
params = {"userId": 1}

# Realizar la petición GET añadiendo el parámetro de userID
respuesta = requests.get(url, params=params)

# Verificar la respuesta del servidor
if respuesta.status_code == 200:
  print("Respuesta recibida correctamente!")
  print(respuesta.json())
else:
  print(f"Fallo en la petición GET: {respuesta.status_code}")

# Ejemplo 3: Petición de tipo POST

# URL donde hacer la petición
url = "https://jsonplaceholder.typicode.com/posts"

# Datos a enviar en la petición
datos = {
    "title": "Ejemplo petición POST",
    "body": "Ejemplo de body de nuestra petición POST",
    "userId": 1
}

# Hacer consulta mediante POST
respuesta = requests.post(url, json=datos)
print("#" * 50)
print("Ejemplo de POST")
# Verificar la respuesta del servidor
if respuesta.status_code == 200:
  print("Respuesta recibida correctamente!")
  print(respuesta.json())
else:
  print(f"Fallo en la petición POST: {respuesta.status_code}")

# Manejo de errores en solicitudes HTTP
print("#" * 50)
print("Imprimiendo errores de peticiones HTTP")
try:
  respuesta = requests.get("https://url-inventada.com")
  respuesta.raise_for_status()
except requests.exceptions.RequestException as error:
  print(f"Ocurrió un error: {error}")

# Ejemplo 4: Añadiendo encabezados a una solicitud

```

```python
# Definir la URL
url = "https://jsonplaceholder.typicode.com/posts"

# Crear los encabezados personalizados
encabezados = {
    "Authorization": "Bearer Nuestro_token",
    "Content-Type": "applicaion/json"
}

print("#" * 50)
print("Haciendo peticiones con encabezados personalizados")
# Realizar la solicitud GET
respuesta = requests.get(url, headers=encabezados)

# Verificar la respuesta del servidor
if respuesta.status_code == 200:
  print("Respuesta recibida correctamente!")
  print(respuesta.json())
else:
  print(f"Fallo en la petición POST: {respuesta.status_code}")

# Ejemplo 5: Clase envoltorio para manejar peticiones

print("#" * 50)
print("Creando un envoltorio")

# Clase que envuelve una petición a la API
class APIEnvoltorio:
  # Definir una URL base
  URL = "https://jsonplaceholder.typicode.com"

  @staticmethod
  def peticion_post(userID):
    # Construir la URL completa
    respuesta = requests.get(f"{APIEnvoltorio.URL}/posts/{userID}")
    if respuesta.status_code == 200:
      print("Respuesta recibida correctamente!")
      print(respuesta.json())
    else:
      print(f"Fallo en la petición POST: {respuesta.status_code}")

APIEnvoltorio.peticion_post(1)
APIEnvoltorio.peticion_post(2)
APIEnvoltorio.peticion_post(3)

```

En el primer ejemplo se realiza una solicitud GET para obtener un recurso específico. En el segundo, se usan parámetros en la URL (params) para filtrar los resultados. El tercer ejemplo muestra cómo hacer una petición POST enviando un diccionario como cuerpo de la solicitud (json=datos).

También se incluye un bloque try/except para manejar errores al hacer peticiones, lo que evita que el programa se detenga si ocurre un fallo. En el cuarto ejemplo se añaden encabezados personalizados a la solicitud con el parámetro headers.

Por último, se define una clase como envoltorio que centraliza y reutiliza la lógica de las peticiones, facilitando el mantenimiento del código al trabajar con múltiples endpoints de una API.

Multiprocesamiento en Python

¿Qué es el Multiprocesamiento?

El multiprocesamiento es una técnica que permite ejecutar varias tareas simultáneamente, distribuyendo la carga de trabajo entre múltiples procesos que pueden ejecutarse en diferentes núcleos de la CPU. Cada proceso tiene su propio espacio de memoria, lo que lo hace independiente de otros procesos y evita problemas como la competencia por recursos que ocurre en multihilos (threading).

En Python, el multiprocesamiento es particularmente útil para tareas intensivas de CPU, donde el rendimiento se puede mejorar significativamente aprovechando todos los núcleos disponibles en un sistema.

¿Para qué se utiliza el Multiprocesamiento?

El multiprocesamiento se usa para:

1. Aumentar la velocidad de ejecución: Dividir un problema en subproblemas que se puedan resolver al mismo tiempo.
2. Procesamiento intensivo de datos: Tareas como cálculos matemáticos, análisis de datos, simulaciones físicas o computación científica.
3. Reducción de tiempos de espera: Procesar grandes volúmenes de datos o realizar operaciones repetitivas y costosas en términos de tiempo.

Ejemplos de tareas comunes:

- Procesamiento de imágenes o videos.
- Análisis de grandes conjuntos de datos (Big Data).
- Entrenamiento de modelos de Machine Learning.
- Simulación de sistemas complejos.

¿Cómo funciona el Multiprocesamiento en Python?

El módulo multiprocessing de Python ofrece herramientas para crear y gestionar procesos de manera sencilla. A continuación, se explican sus conceptos clave:

1. Procesos

Un proceso es una instancia de un programa en ejecución. En Python, se puede crear un proceso con la clase multiprocessing.Process. Cada proceso tiene su propio espacio de memoria y se ejecuta independientemente.

2. Espacio de memoria independiente

Los procesos no comparten memoria, lo que los hace más seguros para tareas paralelas, pero requiere mecanismos como Queue o Pipe para intercambiar datos entre procesos.

3. Multiprocesamiento vs Multihilos

- Multiprocesamiento: Cada proceso tiene su propia memoria y se ejecuta en núcleos separados. Es ideal para tareas intensivas de CPU.
- Multihilos (Threading): Los hilos comparten el mismo espacio de memoria dentro de un único proceso. Es más adecuado para tareas intensivas de entrada/salida (I/O), como leer archivos o consultar APIs.

4. Gestión de procesos

El método Process permite crear procesos individuales, pero para tareas repetitivas o basadas en listas, el uso de Pool simplifica la gestión.

5. Colas y tuberías (Queue y Pipe)

Se utilizan para la comunicación entre procesos. Las colas permiten enviar datos de un proceso a otro de forma segura.

6. Protección if name == "main":

Es esencial en Python al usar multiprocessing para evitar que los procesos hijos ejecuten accidentalmente el código principal al ser creados.

```
 1. import multiprocessing
 2. import time
 3. import os
 4. import random
 5.
 6. # Función auxiliar para simular unos cálculos
 7. def proceso_paralelo(dato, cola):
 8.   try:
 9.     print(f"Proceso {os.getpid()} procesando el dato: {dato}")
10.     time.sleep(random.uniform(0.5, 2.0))  # Simular un tiempo variable de
procesamiento
11.     resultado = sum(z**3 for z in range(dato))  # Calcular la suma de los cubos desde
0 hasta nuestro valor de dato
12.     cola.put((dato, resultado))  # Guardar resultado dentro de la cola
13.     print(f"Proceso {os.getpid()} ha terminado con éxito el dato: {dato}")
14.   except Exception as error:
15.     print(f"Ocurrió un error: {error}")
16.     cola.put((dato, f"Error: {error}"))
17.
18. def main():
19.   # Crear una cola para almacenar los resultados de nuestros procesos
20.   cola_resultados = multiprocessing.Queue()
21.
22.   # Lista ficticia de datos a procesar
23.   lista = [5, 10, 15, 20, 25, 30]
24.
25.   # Lista para almacenar los procesos creados
26.   procesos = []
27.
28.   # Iniciamos un contador
29.   print("Iniciando el procesamiento en paralelo de nuestro código")
30.   timepo_inicio = time.time()
31.
32.   # Crear procesos para procesar cada dato en nuestra lista
33.   for dato in lista:
34.     proceso = multiprocessing.Process(target=proceso_paralelo, args=(dato,
cola_resultados))
35.     procesos.append(proceso)
36.     proceso.start()
37.
38.   # Esperar a que todos los procesos terminen
```

```
    for proceso in procesos:
      proceso.join()

    # Recopilar los resultados
    resultados = []
    while not cola_resultados.empty():
      resultados.append(cola_resultados.get())

    # Terminamos el contador
    tiempo_final = time.time()

    # Mostrar por pantalla los datos analizados
    for dato, resultado in resultados:
      print(f"Dato: {dato} tenemos un resultado de: {resultado}")

    # Mostramos el tiempo total de ejecución
    print(f"Tiempo total de ejecucion: {tiempo_final - timepo_inicio:.2f} segundos")

if __name__ == "__main__":
    main()

```

Este programa simula el procesamiento de una lista de datos mediante múltiples procesos ejecutados en paralelo. Cada proceso ejecuta la función proceso_paralelo, que simula un cálculo costoso (suma de cubos) y almacena el resultado en una cola compartida (multiprocessing.Queue).

La función main() coordina la ejecución: crea los procesos, los inicia, espera su finalización con join(), y luego recoge los resultados de la cola. Finalmente, se muestra el resultado de cada dato procesado y el tiempo total de ejecución.

Este enfoque permite ejecutar varias tareas al mismo tiempo, mejorando el rendimiento en sistemas con varios núcleos.

¿Qué es el Multithreading?

El multithreading es una técnica de programación que permite ejecutar múltiples hilos (threads) de manera concurrente dentro de un mismo proceso. Un hilo es la unidad más pequeña de ejecución que puede ser gestionada de forma independiente por el sistema operativo. Esta técnica se utiliza para mejorar la eficiencia y la capacidad de respuesta de los programas, especialmente en tareas que implican operaciones de entrada/salida (I/O).

En Python, el multithreading se implementa mediante el módulo threading. Sin embargo, debido a la presencia del Global Interpreter Lock (GIL), los hilos en Python no pueden ejecutar código Python puro en paralelo dentro de un mismo proceso. Esto significa que el multithreading es más útil en tareas que dependen de la espera (como operaciones de red o escritura/lectura de archivos) que en tareas de uso intensivo de CPU.

Conceptos clave del multithreading en Python

1. **Hilo (Thread)**
   Un hilo es una secuencia independiente de instrucciones que se ejecuta dentro

del contexto de un programa principal. Varios hilos comparten los mismos recursos (memoria, variables, etc.) del proceso.

2. **Concurrente pero no paralelo**
   En Python, debido al GIL, los hilos no ejecutan tareas realmente en paralelo cuando están utilizando código Python puro. En lugar de eso, se alternan en el tiempo, proporcionando concurrencia.

3. **Ámbito de uso**
   El multithreading es ideal para operaciones que implican esperar por recursos externos, como leer o escribir en archivos, solicitudes a bases de datos o llamadas a APIs.

4. **Global Interpreter Lock (GIL)**
   El GIL es un mecanismo en el intérprete de Python que garantiza que solo un hilo pueda ejecutar código Python a la vez. Esto limita la efectividad del multithreading para tareas que consumen mucha CPU.

Uso común del multithreading

El multithreading es especialmente útil en situaciones como:

- Operaciones de red: Manejo de múltiples conexiones de clientes en un servidor.
- Web scraping: Descarga simultánea de datos desde múltiples páginas web.
- Entrada/Salida (I/O): Lectura y escritura de archivos o bases de datos en paralelo para mejorar el rendimiento.

Consideraciones importantes

1. **Seguridad de los hilos (Thread Safety)**
   Cuando varios hilos comparten recursos como variables globales, es necesario evitar conflictos mediante mecanismos de sincronización, como los bloqueos (locks).

2. **Limitaciones del GIL**
   Si bien el multithreading puede mejorar la concurrencia en tareas de I/O, no es efectivo para tareas intensivas en cálculos. Para estas, se recomienda usar el módulo multiprocessing, que permite ejecutar procesos independientes en paralelo.

3. **Sincronización de hilos**
   Es fundamental coordinar los hilos para evitar problemas como condiciones de carrera (race conditions), donde dos o más hilos acceden y modifican datos al mismo tiempo.

```
1. # Importar las librerías
2. import threading
3. import time
4.
5. def tarea_simple(nombre, delay, repeticiones):
```

```
  for index in range(repeticiones):
    time.sleep(delay)
    print(f"{nombre} está pasando por la iteración número: {index}")

# Función para calcular el total mientras otro hilo está trabajando
def calcular_total(limite):
  total = 0
  for i in range(limite):
    total += i
    time.sleep(0.1)
  print(f"El total calculado hasta {limite} es de: {total}")

hilo_1 = threading.Thread(target=tarea_simple, args=("Hilo_1", 1, 5))
hilo_2 = threading.Thread(target=tarea_simple, args=("Hilo_2", 2, 3))
hilo_3 = threading.Thread(target=calcular_total, args=(10,))

# Iniciar los hilos
hilo_1.start()
hilo_2.start()
hilo_3.start()

# Esperar a que todos los hilos finalicen
hilo_1.join()
hilo_2.join()
hilo_3.join()

```

Mensaje final una vez todos los hilos han terminado

print("Final ejercicio, todos los hilos han terminado")

En este ejemplo se crean tres hilos (hilo_1, hilo_2, hilo_3) utilizando la clase Thread de la librería threading. Cada hilo ejecuta una función diferente en paralelo:

tarea_simple: muestra mensajes con un retardo entre cada iteración.

calcular_total: calcula la suma de números hasta un límite dado.

Los hilos se inician con .start() y el programa principal espera a que terminen con .join(). Esto asegura que el mensaje final se imprima solo cuando todos los hilos han completado su ejecución.

El uso de hilos permite realizar varias tareas al mismo tiempo dentro del mismo proceso, mejorando la eficiencia en ciertas situaciones como esperar respuestas o ejecutar múltiples procesos ligeros en paralelo.