Entrada y salida de datos en Python

Objetivo

El objetivo particular de este módulo es enseñar a los estudiantes cómo trabajar con archivos y datos en Python. Esto incluye la lectura y escritura de archivos de texto y binarios, así como la serialización y deserialización de objetos. Los conceptos y técnicas presentadas en este módulo son fundamentales para la manipulación de datos y la interacción con sistemas de archivos en el desarrollo de aplicaciones de Python.

Al finalizar este módulo, los estudiantes deberían ser capaces de:

1. Leer y escribir archivos de texto en Python, utilizando diferentes modos de apertura y técnicas de lectura y escritura.
2. Leer y escribir archivos binarios, útil para trabajar con archivos que no son de texto, como imágenes o archivos comprimidos.
3. Utilizar el módulo `pickle` para serializar y deserializar objetos en Python, lo que permite guardar y recuperar el estado de objetos entre sesiones de ejecución de programas.
4. Trabajar con archivos JSON para almacenar y recuperar datos estructurados, lo que facilita la interoperabilidad con otros sistemas y lenguajes de programación.

Este módulo proporciona a los estudiantes una base sólida en el manejo de archivos y datos en Python, lo que les permitirá abordar una amplia variedad de problemas en el desarrollo de aplicaciones y análisis de datos.

Datos en consola

input()

La función `input()` permite obtener datos ingresados por el usuario en la consola. La función puede recibir un argumento que será el mensaje mostrado al usuario.

```
nombre = input("Ingrese su nombre: ")
print("Hola,", nombre)
```

Ejemplo:

```
numero = int(input("Ingrese un número entero: "))
print("El número ingresado es", numero)
```

print()

La función `print()` permite mostrar información en la consola. Puede recibir múltiples argumentos separados por comas, que serán concatenados automáticamente.

```
nombre = "Juan"
edad = 30
print("Mi nombre es", nombre, "y tengo", edad, "años")
```

Ejemplo:

```
x = 5
y = 3
suma = x + y
print(x, "+", y, "=", suma)
```

Archivos

Python facilita el manejo de archivos, tanto para leer como para escribir en ellos. Se pueden utilizar los modos “r” (lectura), “w” (escritura), “a” (añadir) y “x” (creación exclusiva).

Lectura de archivos

```
with open("archivo.txt", "r") as archivo:
    contenido = archivo.read()
    print(contenido)
```

Ejemplo:

```
with open("numeros.txt", "r") as archivo:
    numeros = [int(line.strip()) for line in archivo.readlines()]
    print("Números:", numeros)
```

Escritura de archivos

```
with open("nuevo_archivo.txt", "w") as archivo:
    archivo.write("Hola, mundo!")
```

Ejemplo:

```
nombres = ["Ana", "Carlos", "Beatriz"]
with open("nombres.txt", "w") as archivo:
    for nombre in nombres:
        archivo.write(nombre + "\n")
```

Añadir contenido a un archivo

```
with open("archivo_existente.txt", "a") as archivo:
    archivo.write("Esta línea se añadirá al final del archivo.\n")
```

Ejemplo:

```
nuevos_nombres = ["David", "Elena"]
with open("nombres.txt", "a") as archivo:
    for nombre in nuevos_nombres:
        archivo.write(nombre + "\n")
```

Lectura de archivos línea por línea

En algunos casos, es útil leer un archivo línea por línea en lugar de cargar todo el contenido en memoria. Esto se puede hacer con un bucle `for`:

```
with open("archivo.txt", "r") as archivo:
    for line in archivo:
        print(line.strip())
```

Ejemplo:

```
with open("palabras.txt", "r") as archivo:
    for linea in archivo:
        palabra = linea.strip()
        print("Palabra:", palabra)
```

Operaciones con archivos en modo binario

Para leer o escribir archivos en modo binario, se debe agregar la letra “b” al modo de apertura del archivo. Esto es útil cuando se trabaja con archivos que no son de texto, como imágenes o archivos comprimidos.

Lectura en modo binario:

```
with open("imagen.jpg", "rb") as archivo:
    contenido = archivo.read()
    print("Contenido en bytes:", contenido)
```

Escritura en modo binario:

```
datos_binarios = b'\x48\x6f\x6c\x61\x20\x6d\x75\x6e\x64\x6f\x21'
with open("archivo_binario.bin", "wb") as archivo:
    archivo.write(datos_binarios)
```

Formateo de cadenas de texto

Python proporciona diferentes maneras de formatear cadenas de texto para facilitar su lectura y presentación.

Método `format()`

El método `format()` permite reemplazar marcadores de posición en una cadena de texto con valores específicos:

```
nombre = "Juan"
edad = 30
frase = "Mi nombre es {} y tengo {} años".format(nombre, edad)
print(frase)
```

F-strings

Las F-strings (cadenas con formato) son una característica de Python 3.6 en adelante que permite incluir expresiones dentro de las cadenas de texto, utilizando llaves y un prefijo “f”:

```
nombre = "Juan"
edad = 30
frase = f"Mi nombre es {nombre} y tengo {edad} años"
print(frase)
```

Ejemplo:

```
x = 5
y = 3
suma = x + y
print(f"{x} + {y} = {suma}")
```

Uso de archivos temporales

En algunos casos, es útil trabajar con archivos temporales que se eliminan automáticamente una vez que se cierra el archivo. Python proporciona el módulo `tempfile` para manejar este tipo de archivos.

```
import tempfile

with tempfile.TemporaryFile(mode="w+t") as archivo_temporal:
    archivo_temporal.write("Esta es una línea en un archivo temporal.")
    archivo_temporal.seek(0)
    contenido = archivo_temporal.read()
    print("Contenido del archivo temporal:", contenido)
```

Conversión de tipos de datos en entrada y salida

A menudo, es necesario convertir tipos de datos al leer o escribir información. Por ejemplo, al leer números de un archivo de texto, estos se encuentran en formato de cadena de caracteres. Es necesario convertirlos a números antes de realizar operaciones matemáticas.

Ejemplo:

```
with open("precios.txt", "r") as archivo:
    precios = [float(line.strip()) for line in archivo.readlines()]

total = sum(precios)
print(f"El total de los precios es: {total}")
```

Funciones de entrada y salida personalizadas

Para facilitar la reutilización de código y mejorar la legibilidad de los programas, es posible crear funciones personalizadas para operaciones de entrada y salida de datos.

Ejemplo:

```
def leer_numeros(archivo):
    with open(archivo, "r") as f:
        numeros = [int(line.strip()) for line in f.readlines()]
    return numeros

def guardar_numeros(archivo, numeros):
    with open(archivo, "w") as f:
        for numero in numeros:
            f.write(f"{numero}\n")

numeros = leer_numeros("numeros.txt")
numeros_ordenados = sorted(numeros)
guardar_numeros("numeros_ordenados.txt", numeros_ordenados)
```

Con estas notas y ejemplos, los estudiantes podrán comprender y practicar el manejo de entrada y salida de datos en Python, incluyendo el uso de funciones `input()` y `print()`, operaciones con archivos de texto y binarios, manejo de excepciones y formateo de cadenas de texto.

Codificación y decodificación de caracteres

Cuando se trabaja con archivos de texto, es posible que se encuentren caracteres especiales o acentuados que no sean parte del conjunto de caracteres ASCII. Python utiliza el estándar Unicode para representar estos caracteres y es compatible con varias codificaciones de caracteres, como UTF-8, UTF-16 y UTF-32.

Codificación

La codificación es el proceso de convertir una cadena de caracteres en una secuencia de bytes. En Python, se puede codificar una cadena de caracteres utilizando el método `encode()` y especificando la codificación deseada:

```
texto = "¡Hola, mundo!"
texto_utf8 = texto.encode("utf-8")
print("Texto codificado en UTF-8:", texto_utf8)
```

Decodificación

La decodificación es el proceso de convertir una secuencia de bytes en una cadena de caracteres. En Python, se puede decodificar una secuencia de bytes utilizando el método `decode()` y especificando la codificación utilizada:

```
bytes_utf8 = b'\xc2\xa1Hola, mundo!'
texto_decodificado = bytes_utf8.decode("utf-8")
print("Texto decodificado:", texto_decodificado)
```

Lectura y escritura de archivos con codificación específica

Al leer o escribir archivos de texto, es posible especificar la codificación a utilizar utilizando el argumento `encoding` en la función `open()`:

```
# Escritura de un archivo con codificación UTF-8
with open("archivo_utf8.txt", "w", encoding="utf-8") as archivo:
    archivo.write("¡Hola, mundo!")

# Lectura de un archivo con codificación UTF-8
with open("archivo_utf8.txt", "r", encoding="utf-8") as archivo:
    contenido = archivo.read()
    print("Contenido del archivo UTF-8:", contenido)
```

Compresión y descompresión de archivos

Python ofrece módulos para trabajar con archivos comprimidos, como `gzip`, `bz2` y `zipfile`. Estos módulos permiten comprimir y descomprimir archivos fácilmente.

Compresión con gzip

El siguiente ejemplo muestra cómo comprimir un archivo utilizando el módulo `gzip`:

```
import gzip

with open("archivo.txt", "rb") as f_in, gzip.open("archivo.txt.gz", "wb") as f_out:
    f_out.writelines(f_in)
```

Descompresión con gzip

El siguiente ejemplo muestra cómo descomprimir un archivo utilizando el módulo `gzip`:

```
import gzip

with gzip.open("archivo.txt.gz", "rb") as f_in, open("archivo_descomprimido.txt", "wb") as f_c
    f_out.writelines(f_in)
```

Serialización y deserialización de datos

La serialización es el proceso de convertir un objeto en un formato que se pueda almacenar o transmitir fácilmente. Python proporciona el módulo `pickle` para serializar y deserializar objetos.

Serialización de un objeto

El siguiente ejemplo muestra cómo serializar un objeto utilizando el módulo `pickle`:

```
import pickle

datos = {"nombre": "Juan", "edad": 30, "profesion": "Ingeniero"}

with open("datos.pkl", "wb") as archivo:
    pickle.dump(datos, archivo)
```

Deserialización de un objeto

El siguiente ejemplo muestra cómo deserializar un objeto utilizando el módulo `pickle`:

```
import pickle

with open("datos.pkl", "rb") as archivo:
    datos_cargados = pickle.load(archivo)

print("Datos deserializados:", datos_cargados)
```

Trabajo con archivos JSON

JSON (JavaScript Object Notation) es un formato de intercambio de datos ligero y fácil de leer y escribir. Python proporciona el módulo `json` para trabajar con archivos JSON.

Escritura de un archivo JSON

El siguiente ejemplo muestra cómo escribir un archivo JSON utilizando el módulo `json`:

```python
import json

datos = {
    "nombre": "Juan",
    "edad": 30,
    "profesion": "Ingeniero"
}

with open("datos.json", "w") as archivo:
    json.dump(datos, archivo)
```

Lectura de un archivo JSON

El siguiente ejemplo muestra cómo leer un archivo JSON utilizando el módulo `json`:

```python
import json

with open("datos.json", "r") as archivo:
    datos_cargados = json.load(archivo)

print("Datos cargados desde JSON:", datos_cargados)
```

Manejo de errores al trabajar con archivos

Es importante manejar los errores que puedan surgir al trabajar con archivos, como errores de permisos, archivos inexistentes o problemas de lectura y escritura. Python utiliza excepciones para manejar errores en tiempo de ejecución. Las excepciones más comunes relacionadas con el manejo de archivos son `FileNotFoundError`, `PermissionError` e `IOError`.

Ejemplo de manejo de excepciones al abrir un archivo

El siguiente ejemplo muestra cómo manejar errores al intentar abrir un archivo:

```python
try:
    with open("archivo_inexistente.txt", "r") as archivo:
        contenido = archivo.read()
except FileNotFoundError:
    print("El archivo no se encuentra.")
except PermissionError:
    print("No se tienen permisos para leer el archivo.")
except IOError:
    print("Ocurrió un error de entrada/salida al leer el archivo.")
```

Este ejemplo utiliza un bloque `try` para intentar leer el contenido de un archivo. Si se produce un error, el bloque `except` correspondiente manejará la excepción y mostrará un mensaje apropiado.

Resumen y conclusiones

En este módulo, hemos aprendido sobre el manejo de archivos en Python, incluyendo la lectura y escritura de archivos de texto y binarios, la serialización y deserialización de objetos utilizando el módulo `pickle`, y el trabajo con archivos JSON utilizando el módulo `json`. Además, hemos discutido la importancia del manejo de errores y excepciones relacionadas con el manejo de archivos.

Hemos visto cómo utilizar la función `open()` junto con los modos de apertura de archivos (`r`, `w`, `a`, `x`, `b`) para abrir archivos y realizar diversas operaciones en ellos. También hemos aprendido sobre el uso del administrador de contexto `with` para garantizar que los archivos se cierren correctamente después de su uso.

La serialización y deserialización de objetos nos permite almacenar y recuperar estructuras de datos en archivos, mientras que el trabajo con archivos JSON nos permite intercambiar datos en un formato más universal y fácilmente legible.

Manejar errores y excepciones es esencial para garantizar que nuestros programas sean sólidos y resistentes a problemas comunes, como la falta de permisos o la inexistencia de archivos.

En resumen, el manejo de archivos es una habilidad fundamental para cualquier desarrollador de Python, ya que es esencial para la manipulación y almacenamiento de datos, la comunicación entre aplicaciones y la construcción de programas robustos y resistentes a errores.

Perspectivas de aprendizaje

Con este conocimiento sobre el manejo de archivos en Python, puede continuar explorando y profundizando en temas relacionados, como:

- Trabajo con archivos CSV y otros formatos de datos tabulares.
- Uso de bases de datos y módulos de Python para la gestión de datos más avanzada.
- Comprensión y aplicación de técnicas de codificación y compresión de datos.
- Mejora de las habilidades de manejo de excepciones y depuración en Python.

Además, es importante seguir practicando y experimentando con diferentes tipos de archivos y operaciones para familiarizarse con las diversas técnicas y mejorar sus habilidades en el manejo de archivos en Python.