**﻿﻿Módulo 5.2**

**Bibliotecas en Python**

**almacenamiento de datos en archivos externos**

* + **csv**
  + **json**
  + **pickle**

**CSV (Comma-Separated Values)**

Un archivo CSV es un formato de archivo de texto que se utiliza para almacenar datos en forma de tabla, en la cual las columnas están separadas por comas y las filas por saltos de línea. Cada fila representa un registro o conjunto de datos relacionados, y cada columna representa un atributo o campo de información del registro.

Los valores en cada fila están separados por comas (o algún otro delimitador, como punto y coma o tabulación).

Al permitir almacenar datos tabulares en forma de texto plano, archivo CSV, son muy utilizados para el intercambio de datos entre diferentes sistemas y aplicaciones, se pueden leer o escribir en un note – bloc de notas o en una planilla de calculo (Excel, calc,etc), bases de datos y otros.

Mas adelante veremos que por intermedio de pandas los archivos CSV son muy útiles para trabajar con grandes cantidades de datos ya que son fáciles de manipular mediante scripts y programas automatizados.

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | import csv  def operaciones\_sobre\_csv(archivo,entrada="",modo="w"):  if modo.startswith("w"):  with open (archivo, mode=modo, newline='') as io\_obj:  print(entrada[0])  if len(entrada)>1 and isinstance(entrada[0],dict):  print(entrada[0].keys())  escritor\_csv = csv.DictWriter(io\_obj, fieldnames=list(entrada[0].keys()))  escritor\_csv.writeheader()  print ("desde diccionarios")  for cant,cada\_entrada in enumerate(entrada):  print (cada\_entrada)  escritor\_csv.writerow(cada\_entrada)  elif len(entrada)>1 and isinstance(entrada[0],list):  print ("Desde listas")  escritor\_csv = csv.writer(io\_obj)  for cant,cada\_entrada in enumerate(entrada):  escritor\_csv.writerow(cada\_entrada)  print (f"se han escrito {cant+1} registros\n\n\n")#inicia en cero  elif modo.startswith("r") or "+" in modo:  salida={}  with open (archivo, mode=modo, newline='') as io\_obj:  lector\_csv = csv.DictReader(io\_obj)  for item,registro in enumerate(lector\_csv):  if isinstance(registro,dict):  for clave, valor in registro.items():  print (f"la clave '{clave}' posee el valor '{valor}'")  salida[str(item)]=registro  return salida  datos\_tabulares\_listas=[  ['Nombre', 'Apellido', 'Edad', 'Estudia'],  ['Juan', 'García','20','Back end'],  ['Pedro','González', '22','Analisis de datos'],  ['Luciana', 'Rodríguez','24','Front end'],  ['Analia', 'Perez', '26','Juegos'],  ]  datos\_tabulares\_dict=[  {'Nombre':'Juan','Apellido':'García', 'Edad':'20', 'Estudia':'Back end'},  {'Nombre':'Pedro', 'Apellido':'González', 'Edad':'22', 'Estudia':'Analisis de datos'},  {'Nombre':'Luciana','Apellido':'Rodríguez', 'Edad':'24', 'Estudia':'Front end'},  {'Nombre':'Analia', 'Apellido':'Perez', 'Edad':'26', 'Estudia':'Juegos'},  ]  operaciones\_sobre\_csv("Mineria.csv",datos\_tabulares\_listas,"w")  datos=operaciones\_sobre\_csv("Mineria.csv",modo="r")  print (f"{datos=}")  print("\*"\*50)  operaciones\_sobre\_csv("Mineria.csv",datos\_tabulares\_dict,"w")  datos=operaciones\_sobre\_csv("Mineria.csv",modo="r")  print (f"{datos=}") |   ·Salida esperada por consola   |  | | --- | | Verificar salidas |   · |

·

La biblioteca csv de Python proporciona varios métodos y funciones útiles para trabajar con archivos CSV.

Método reader

csv.reader(file, dialect='excel', \*\*kwargs):

El primer parámetro es el archivo CSV que se va a leer, el segundo puede ser un dialecto o pasar los distintos parametros por clave valor.

* ****Dialecto excel**** : Este es el dialecto CSV predeterminado que usa Microsoft Excel. Utiliza comas como separador de campos y comillas dobles para delimitar los campos que contienen caracteres especiales como comas o saltos de línea.
* ****Dialecto unix:**** Este dialecto es similar al dialecto 'excel', pero utiliza una sola comilla simple para delimitar los campos en lugar de comillas dobles. También utiliza un salto de línea (\n) como delimitador de registro en lugar de la combinación de retorno de carro y salto de línea (\r\n) utilizada en Windows.
* ****Dialecto personalizado****: También puedes definir tu propio dialecto CSV personalizado utilizando la clase csv.Dialect. Aquí hay un ejemplo de cómo definir un dialecto personalizado que utiliza un punto y coma como separador de campos y comillas simples para delimitar los campos:

Los parametros que se le pueden pasar son :

* + - ****delimiter****: El carácter utilizado para separar los campos en el archivo CSV. El valor predeterminado es una coma (',').
    - ****quotechar****: El carácter utilizado para delimitar los campos que contienen caracteres especiales como el separador de campos o el salto de línea. El valor predeterminado es una comilla doble (' " ').
    - ****escapechar****: El carácter utilizado para escapar las comillas delimitadoras en un campo. Por defecto, el carácter de escape es la comilla doble.
    - ****skipinitialspace****: Si se establece en True, los espacios en blanco al comienzo de un campo se ignoran al leer el archivo CSV.
    - ****ineterminator****: El carácter utilizado para separar las líneas en el archivo CSV. Por defecto, se utiliza la secuencia de caracteres de retorno de carro y salto de línea ('\r\n').
    - ****quoting****: Controla cuándo se utilizan las comillas delimitadoras en los campos. Puede tomar uno de los siguientes valores:
* csv.QUOTE\_ALL: Todas las cadenas de campos se delimitan con comillas.
* csv.QUOTE\_NONNUMERIC: Solo se delimitan las cadenas de campos que no son numéricas.
* csv.QUOTE\_NONE: No se utilizan comillas delimitadoras.

DictWriter

Crea un objeto escritor de CSV que permite escribir un archivo CSV como un diccionario en lugar de una lista. El parámetro fieldnames se utiliza para especificar los nombres de las columnas en el archivo CSV.

**Serialización de archivos (convertir en una secuencia de bytes)**

La serialización de archivos es el proceso de convertir una estructura de datos o un objeto en una secuencia de bytes para que pueda ser almacenado en un archivo o transmitido a través de una red. La serialización permite guardar el estado de un objeto de manera persistente, lo que significa que se puede recuperar posteriormente y restaurar en su forma original.

La serialización es útil cuando necesitas almacenar objetos complejos en disco, transferir datos entre diferentes plataformas o comunicarte con sistemas distribuidos. Al serializar un objeto, se guarda su estado actual, incluyendo los valores de sus atributos y la estructura de los datos.

La serialización puede realizarse en diferentes formatos, como JSON (JavaScript Object Notation), XML (eXtensible Markup Language), protocol buffers, entre otros. Estos formatos definen reglas y convenciones para convertir los datos en una representación legible por máquina.

En Python, el módulo por excelencia es pickle proporciona funcionalidad para serializar y deserializar objetos en formato binario. También existen otras bibliotecas populares de serialización como json y xml.etree.ElementTree que permiten la serialización en formatos JSON y XML, respectivamente.

**JSON (JavaScript Object Notation)**

**pip install json**

Los archivos json son utilizado para almacenar y transportar datos estructurados en forma de texto plano. JSON es ampliamente utilizado debido a su simplicidad y facilidad de lectura tanto para humanos como para máquinas.

En un archivo JSON, los datos se organizan en forma muy parecida a los diccionarios en Python, objetos y arrays, que son estructuras de datos que pueden contener valores simples, otros objetos JSON o arrays JSON.

La biblioteca json en Python es una herramienta esencial para trabajar con JSON. Proporciona funciones para codificar y decodificar datos en formato JSON. Algunas de las funciones más comunes incluyen:

json.dumps(): Codifica datos de Python en formato JSON, devolviendo una cadena JSON.

json.loads(): Decodifica una cadena JSON en objetos Python.

json.dump(): Escribe datos de Python en formato JSON en un archivo.

json.load(): Lee datos JSON de un archivo y los convierte en objetos Python.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **json.dump vs json.dumps**  DUMP: Graba un objeto Python a un archivo JSON  DUMPS: Convierte un objeto Python a un objeto JSON   |  |  | | --- | --- | | Python | JSON | | list, tuple  str  int  float  True  False  dict  None | array  string  number (int)  number (real)  true  false  object  null |   . |

·

|  |
| --- |
| **Indent**  Define el número de niveles de indentación (espacios) que se aplicarán a cada nivel en la estructura JSON.  **sort\_keys**  Ordena las claves del resultado en el objeto JSON.  **separators=(". ", " = ")**  Cambia los separadores por defecto en el formato JSON. Aquí, el primer elemento reemplaza la coma , entre ítems y el segundo reemplaza los dos puntos : entre clave y valor. Sintaxis:  json.dumps(  obj,   \*,  skipkeys=False,  ensure\_ascii=True,  check\_circular=True,   allow\_nan=True,  cls=None,  indent=None,  separators=None,   default=None,   sort\_keys=False,   \*\*kw  )  **Parámetros:**  **obj:** El objeto que será serializado en un flujo de texto con formato JSON. **skipkeys:** Si skipkeys es verdadero (por defecto: Falso), entonces las claves del diccionario que no sean de un tipo básico (como str, int, float, bool, None) serán omitidas en lugar de generar un TypeError. **ensure\_ascii:** Si ensure\_ascii es verdadero (por defecto), el resultado garantizará que todos los caracteres no ASCII se escapen. Si es falso, estos caracteres se mostrarán tal cual. **check\_circular:** Si check\_circular es falso (por defecto: Verdadero), se omitirá la verificación de referencias circulares en tipos de contenedores, lo que podría generar un OverflowError (o peor). **allow\_nan:** Si allow\_nan es falso (por defecto: Verdadero), intentar serializar valores flotantes fuera de rango (nan, inf, -inf) generará un ValueError en cumplimiento estricto de la especificación JSON. Si es verdadero, se usarán los equivalentes de JavaScript (NaN, Infinity, -Infinity). **indent:** Si indent es un entero no negativo o una cadena, los elementos de los arreglos y los miembros de los objetos en JSON se imprimirán con esa cantidad de espacios o con esa cadena como indentación por nivel. Un nivel de indentación de 0, negativo o "" solo insertará saltos de línea. None (el valor por defecto) selecciona la representación más compacta. Usar un número entero positivo indica la cantidad de espacios por nivel. Si indent es una cadena (como \t), esa cadena se usará para indentar cada nivel. **separators:** Si se especifica, separators debe ser una tupla (separador\_item, separador\_clave). El valor por defecto es (",", ":") si indent es None y (",", ":") de otra manera. Para obtener la representación JSON más compacta, especifica la eliminación de espacios en blanco. **default:** Si se especifica, default debe ser una función que se llamará para los objetos que no se puedan serializar de otra manera. Debe devolver una versión codificable en JSON del objeto o lanzar un TypeError. Si no se especifica, se lanza un TypeError. **sort\_keys:** Si sort\_keys es verdadero (por defecto: Falso), entonces la salida de los diccionarios se ordenará por clave. |

·

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **json.dump vs json.dumps**  LOAD: Lee un archivo JSON a un objeto Python  LOADS: Convierte un objeto JSON a uno Python   |  |  | | --- | --- | | JSON | Python | | array  string  number (int)  number (real)  true  false  object  null | list, tuple  str  int  float  True  False  dict  None |   . |

JSON, o JavaScript Object Notation, es un formato de representación de datos estructurados mediante pares clave-valor. Los datos se organizan en objetos JSON, que son colecciones de pares clave-valor encerrados en llaves {}. Las claves son cadenas que identifican los valores, y estos valores pueden ser de diversos tipos, como números, cadenas, booleanos, matrices e incluso objetos JSON anidados.

|  |
| --- |
| El formato Json no soporta todos los tipos de objetos que podría soportar un diccionario Python por lo que hay que pre-procesarlos antes de enviarlos |

Es importante notar algunas similitudes y diferencias entre los objetos JSON y los diccionarios de Python:

**Similitudes:**

Ambos almacenan datos en pares clave-valor.

Tanto JSON como Python son legibles y escribibles fácilmente para los humanos.

Pueden contener diferentes tipos de datos, como números, cadenas, booleanos, matrices y objetos anidados.

**Diferencias:**

JSON es un formato de intercambio de datos independiente del lenguaje, mientras que los diccionarios son una estructura de datos nativa de Python.

JSON admite principalmente tipos de datos primitivos y estructuras de datos anidadas, mientras que los diccionarios de Python pueden contener cualquier objeto Python como valor, incluyendo estructuras de datos como listas, tuplas y diccionarios anidados.

La representación visual es similar con llaves {}, pero los valores en JSON deben ir siempre entre comillas dobles, mientras que los valores en los diccionarios de Python pueden carecer de comillas o usar comillas simples o dobles.

Los diccionarios de Python son mutables y pueden actualizarse directamente, mientras que los objetos JSON son inmutables, lo que significa que para actualizarlos, se debe crear un nuevo objeto JSON que reemplace el original.

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | import json  dicc\_a\_json={  "leg\_001": {'Nombre':'Juan','Apellido':'García', 'Edad':'20', 'Estudia':'Back end'},  "leg\_002": {'Nombre':'Pedro', 'Apellido':'González', 'Edad':'22', 'Estudia':'Analisis de datos'},  "leg\_003": {'Nombre':'Luciana','Apellido':'Rodríguez', 'Edad':'24', 'Estudia':'Front end'},  "leg\_004": {'Nombre':'Analia', 'Apellido':'Perez', 'Edad':'26', 'Estudia':'Juegos'},  }  print("Objeto dict:\n\t",dicc\_a\_json, type(dicc\_a\_json))  obj\_json = json.dumps(dicc\_a\_json, ensure\_ascii=False, indent=4, sort\_keys=True)  print("Objeto json:\n\t",obj\_json, type(obj\_json))  with open("archivo\_json.json", "w",encoding='utf-8') as io\_json:  json.dump(dicc\_a\_json, io\_json, indent=4, sort\_keys=True)  del dicc\_a\_json  with open("archivo\_json.json", "r", encoding='utf-8') as io\_json:  dicc\_a\_json= json.load(io\_json)  print("Archivo json:\n\t",obj\_json, type(obj\_json)) |   ·Salida esperada por consola   |  | | --- | | Verificar salidas |   · |

·

**PICKLE**

Pickle es un módulo en Python que se utiliza para serializar (convertir en una secuencia de bytes) objetos Python y luego deserializarlos (convertirlos nuevamente en objetos Python) en un formato que pueda almacenarse en archivos o enviarse a través de la red. La serialización es el proceso de convertir un objeto en una representación de bytes que puede almacenarse o transmitirse, y la deserialización es el proceso de reconstruir el objeto a partir de esa representación de bytes.

**Características:**

Soporte para una amplia variedad de objetos: Pickle puede serializar y deserializar una amplia gama de objetos de Python, incluyendo números, cadenas, listas, diccionarios, tuplas, conjuntos y objetos personalizados definidos por el usuario.

Almacenamiento persistente: Pickle se utiliza comúnmente para almacenar datos de manera persistente en archivos. Puedes guardar objetos Python en archivos y luego cargarlos nuevamente más tarde, lo que es útil para guardar estados de aplicaciones o para la persistencia de datos en bases de datos.

Compatibilidad con versiones anteriores: Pickle es capaz de manejar cambios en la estructura de los objetos a lo largo del tiempo, lo que significa que puedes cargar objetos serializados incluso si la definición de clase ha cambiado desde que se serializaron.

**Seguridad:** Pickle no es seguro contra datos maliciosos. No debes cargar datos de fuentes no confiables con el módulo pickle ya que podría ejecutar código malicioso si se carga un objeto comprometido.

El módulo pickle proporciona dos funciones principales para serialización y deserialización:

**pickle.dump**(objeto, archivo):

Esta función toma un objeto Python y un archivo abierto en modo binario y serializa el objeto en el archivo.

**pickle.load**(archivo):

Esta función toma un archivo abierto en modo binario que contiene datos serializados y deserializa esos datos para crear un objeto Python.

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | import pickle  dicc\_a\_pkl={  "leg\_001": {'Nombre':'Juan','Apellido':'García', 'Edad':'20', 'Estudia':'Back end'},  "leg\_002": {'Nombre':'Pedro', 'Apellido':'González', 'Edad':'22', 'Estudia':'Analisis de datos'},  "leg\_003": {'Nombre':'Luciana','Apellido':'Rodríguez', 'Edad':'24', 'Estudia':'Front end'},  "leg\_004": {'Nombre':'Analia', 'Apellido':'Perez', 'Edad':'26', 'Estudia':'Juegos'},  }  with open("archivo\_pkl.pkl", "wb") as io\_pkl:  pickle.dump(dicc\_a\_json, io\_pkl)  del dicc\_a\_pkl  with open("archivo\_pkl.pkl", "rb") as io\_pkl:  dicc\_a\_pkl= pickle.load(io\_pkl)  print("Archivo json:\n\t",dicc\_a\_pkl, type(dicc\_a\_pkl)) |   ·Salida esperada por consola   |  | | --- | | Verificar salidas |   · |

·